A secure and conflict free control platform for Care-O-Bot 4

Mikko Seppälä

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 3.5.2018

Thesis supervisor:

Prof. Ville Kyrki

Thesis advisor:

D.Sc. Roel Pieters

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Mikko Seppälä

Title: A secure and conflict free control platform for Care-O-Bot 4

Date: 3.5.2018 Language: English Number of pages: 7+77

Department of Electrical Engineering and Automation

Professorship: Automation Technology

Supervisor: Prof. Ville Kyrki

Advisor: D.Sc. Roel Pieters

This thesis presents an implementation on how resources can be allocated in robotic applications. The implemented system provides access control for movement commands on robots utilizing the Robot Operating System (ROS) as the control framework. The system ensures that unintentional movements cannot be executed by unauthorized motion controllers. Each controller is isolated from the hardware interface. This is enforced by a firewall monitoring internode ROS connections.

The system also provides control over the robot movement independent from the controllers by stopping or slowing down the movements on the robot. This includes configurable velocity and acceleration limits for the commands relayed to the robot hardware. The independent control enables the system to handle faults by constructing separate, reusable fault monitors, which can stop the movement on the robot.

The developed middleware was configured for Care-O-bot 4 personal servant robot. Example controllers and use cases were constructed to test and demonstrate the capabilities of the system. A simulated robot was used to test hazardous examples, such as extended robot arms colliding with walls during navigation. This was resolved by constructing a fault monitor, which stopped the robot if arms were close to a collision. The fault was resolved by folding the arms. The test was successful as the default ROS navigation node was in control of the robot base movement, but the system modified the relayed velocities and the arms did not collide with the walls.

The middleware was proved on the real robot and the differences between the simulated and real robots were insignificant for normal operation. The system delivered the designed functionality.

Keywords: ROS, automation, firewall, safety, service robotics, personal care robots, access control

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Mikko Seppälä

Työn nimi: Turvallinen ja konfliktivapaa kehitysalusta Care-O-bot 4 robotille

Päivämäärä: 3.5.2018 Kieli: Englanti Sivumäärä: 7+77

Sähkötekniikan ja automaation laitos

Professuuri: Automaatiotekniikka

Työn valvoja: Prof. Ville Kyrki

Työn ohjaaja: TkT Roel Pieters

Tässä diplomityössä toteutetaan ja testataan robotin turvallisuutta parantava järjestelmä. Toteutettu järjestelmä on tarkoitettu ROS ohjelmistolla toteutettuihin robotteihin ja erottaa liikkeitä komentavan ohjelmiston ja robotin toimilaitteita ohjaavat ajurit toisistaan. Se antaa vain yhden ohjelmiston kerrallaan komentaa toimilaitetta ja täten varmistaa ettei epätoivottuja liikkeitä muilta ohjaimilta voida suorittaa samanaikaisesti. Tämän varmistetaan ROS-komponenttien välisten kommunikoinnin lukitsemisella käyttäen palomuuria.

Järjestelmä myös tarjoaa mahdollisuuden muokata liikekomentoja ennen niiden toteutusta ja konfiguroitavissa olevat nopeus- ja kiihtyvyysrajoitukset suojelevat käyttäjiä ja ympäristöä mahdollisilta vaaratilanteilta. Tämä myös mahdollistaa ohjaimista riippumattomat pysäytys- ja hidastustilat, joita voidaan käyttää robottiin kohdistuvien vikatilanteiden hallintaan. Näitä voidaan hyödyntää esimerkiksi käyttämällä erillisiä vika- ja vaaratilanteita tunnistavia komponentteja robotin toiminnan ohjaamiseen.

Kehitetty ohjelmisto konfiguroidaan Care-O-bot 4 palvelurobotille ja sen toimintaa testataan ja demonstroidaan rakentamalla yksinkertaisia kontrollereita ja esimerkkitapauksia. Vaarallisten tilanteiden testaamiseen käytetään simulaatioympäristöä, kuten tapausta jossa robotin kädet on ojennettu sivuille ja ne osuvat navigointiympäristön seinäelementteihin. Rakennettu tarkkailija estää käsien tormäämisen pysäyttämällä robotin. Navigoinnin jatkamiseksi käsi komennetaan robotin rungon lähelle ja robotin annetaan jälleen liikkua. Robotti pysäytettiin ja käden mahdollinen törmäys estettiin, vaikka navigointikomponentti komensi robottia liikkumaan koko toimenpiteen ajan.

Järjestelmä todettiin toimivaksi ja se pystyy suojelemaan robottia, ympäristöä ja robotin kanssa työskenteleviä henkilöitä oikein käytettynä.

Avainsanat: ROS, automaatio, palomuuri, turvallisuus, palvelurobotiikka, henkilökohtaiset robottiapurit, resurssienhallinta

Contents

Abbreviations

Abbreviations

API	Application Programming Interface
CAN	Controller Area Network.
DOF	Degrees of freedom
DWA	Dynamic Window Approach
FW	Firewall
HTML	Hypertext Markup Language
HTTP	Hypertext Transport Protocol
MUX	Multiplexer
NTP	Network Time Protocol
PID	Process Identification (number)
ROS	Robot Operating System
SSH	Secure Shell
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UI	User Interface
VLAN	Virtual Local Area Network
XMLRPC	Extensible Markup Language Remote Procedure Call
YAML	YAML Ain't Markup Language

1 Introduction

The advancements in computing technology have increased the processing power significantly. The transistor count on processors has been increasing as the fabrication methods improve [1]. Moore's law [2] has predicted this trend. The price of processing power has also decreased and more powerful system can be gained for the same amount of money as the technology advances [3]. Although Moore's Law may not fully apply anymore [4] the architectural advancements, miniaturization and power efficiency have enabled more applications to have significant computing power at their disposal. This combined with battery technology producing lighter and volumetrically more efficient batteries [5] have led to a multitude of mobile applications.

One of such applications is mobile robotics, which covers multitude of different platforms and objectives the robots must accomplish. Unlike industrial robots, which are usually programmed to execute a single or a series of preprogrammed tasks in a familiar environment, the mobile robots are working in dynamic surroundings. This requires the robots to behave autonomously as their behaviour is not fixed and they have a decisional layer directing the actions taken by the robot [6]. The research and development in control software with integrated safety features have made interaction with humans easier as the risks involved in human robot interaction (HRI) can be identified and addressed autonomously [7].

The interaction between humans and mobile robots have led to assistive technologies for improving the way of human life. Specifically in applications "which require close human-robot interaction and collaborations, as well as physical human-robot contact" [8] the personal care robots are used. Not all personal care robots are mobile [9], but mobile platforms provide more variety for the tasks the robots can perform. The field is not new as robots assisting humans in their everyday life have been built before. For example the vacuum cleaning robots have emerged as successful commercial applications from multiple manufacturers in the 2010s, but the earliest prototypes for autonomous vacuum cleaners were introduced in 1991 [10].

Using machinery to serve and help humans is nothing new. In the 1920 play "Rossum's Universal Robots" the writer Karel Čapek used the word robot to describe the manufactured, soulless workers. The word was developed from the Czech word "robota" describing forced labour [11]. This gained ground and robots were used to describe numerous biomechanical and mechanical servants in literature and movies. Technology was not ready for such servant in the early 1900s, but later the industrial robots started emerging. These were not serving humans as in the original works and were limited to a set of preprogrammed tasks. The modern personal care robots have started closing the loop. Some even have humanoid or animalistic features to improve the interaction with humans [12]. This thesis focuses on the mobile personal care robots, specifically on the mobile servant robot type of machines.

The multitude of different tasks such a platform can execute is immense. The robot may be directed by a decisional layer, but the movements and sensory perception

are handled by separate processes. Due to the diversity of the tasks, a single process is unlikely to handle all the movements required for accomplishing the functionality. As there are multiple controllers capable of commanding the hardware, there is a possibility that a software error or misconfiguration can lead to situations where the robot is performing unintended motions. This leads to issues as robots sharing common space with humans must ensure that they are not endangering the personnel interacting with the robot [8, 13].

This thesis aims to provide protection against this issue through a middleware isolating the controllers from the hardware. This was to be implemented for a mobile servant robot in possession of the Intelligent robotics group at Aalto University [14]. The robot is Care-O-bot 4 manufactured by Fraunhofer IPA [15] (Fraunhofer-Institut für Produktionstechnik und Automatisierung) and it has a humanoid upper body operating on a wheeled chassis. Multiple actuator options exist for the robot with different drive systems, number of arms and articulating joints for the body. The Care-O-bot 4-8 for the research group is outfitted with two arms, an omnidirectional drive system and a 3 DOF (degrees of freedom) joint for the head. It is running ROS [16] (Robot Operating System) Indigo as the operating system for robotic applications. The system for the controller isolation needed to be implemented for this environment. The existing controllers need to stay relatively unchanged so the middleware cannot be inserted into the controllers themselves. Therefore the external system was requested.

The main goal was to ensure that the controllers are not able interfere with the hardware when they were not allowed to. Secondary goals were added later in the development as the system started to form. It should be easy to use so researchers and other users can incorporate it for their experiments without too much of difficulty. The system should also be able to stop the movements of the robots in case of errors or failures are detected in the robot or the monitored environment and provide a way for resolving these situations.

All in all the system to be designed and implemented should be suitable for the research group. Documentation on how the system works should be presented along examples how the system can be utilized. Code examples should also be supplied so the system can be easily utilized for future experiments.

This thesis is sectioned into 11 parts, which consist of five different categories. The existing development is discussed first before the design of the system is explained. The communication and the operation of the components the system consists of are reviewed before the testing and example use cases are presented. Finally the thesis is concluded by discussing the implemented system and its usability on robotic platforms.

2 Requirements and existing development

The thesis was started by looking into previously developed solutions for ROS, which could be utilized for the Care-O-bot 4 implementation. The basic requirements for the system need to be known for evaluating the feasibility of the existing software components. The different controllers, ROS nodes, needed to be isolated from the hardware and an access control setup was needed for modifying the isolated controllers. These were the critical requirement, which the system must accomplish. Other requirements were added later as features were added during the system development and they are listed in the following section.

2.1 Requirements

1. Maximum of one controller per hardware shall access it at a time.
2. The active controller shall be selectable.
3. The hardware interfaces should be protected from outside influence.
4. The commands to hardware should be limited to configurable maximum values.
5. The hardware should be stoppable independent of the active controllers.
6. The ongoing movements should be able to slowed down independent of the active controllers.
7. The system should not allow unauthorized access modifications.
8. The system should provide recovery methods for faults.
9. The system should be configurable for multiple platforms.
10. The system should support multiple robots on shared environment.

2.2 Existing development

The acquired results for readymade components were not promising as other people had asked the questions about ROS access control before. The closest answer was the usage of firewall on the ROS TCP and UDP connections [17]. Another possible solution was found with SROS. The third Secure ROS, which should not be confused with SROS, was announced later when the implementation for thesis had been started, but the provided features seemed similar to SROS so it was inspected.

2.2.1 SROS

SROS [18] is a security enhancement for ROS and it is proposed as an addition to the ROS API for fixing the security vulnerabilities in ROS. It incorporates TLS

(Transport Layer Security) into the communications used by ROS along other modern cryptography and security measures. TLS is intended to protect the nodes from network threats, even local ones, by encrypting the traffic between ROS components with Public Key Infrastructure (PKI). Encryption ensures that the packets cannot be redirected or replayed to cause harm on the robot by injecting actions into the system. This is achieved by using the ROS client libraries to add the use of TLS transport instead of the normal node communications [19]. This allows the existing codebase to support SROS without modifications [20]. Currently the only supported interface is the TCP transport layer when developed under Python with the rospy [18] client library. The system also provides userspace tools for generating the node key pairs for the encryption and access control policies. Other supporting components are made to distribute these keys on system setup.

Access control is implemented through extensions on the PKI, but it is used for restricting the access the node has. Namespaces and topics the node may subscribe or publish into can be configured. The node may not increase its own privileges on the system as the TLS handshakes on the connections would otherwise fail. Essentially every connection is secured through the TLS and the permissions must match on the key or the connection may not form.

SROS also goes on the kernel level restricting the process, the ROS node, access to resources using the Linux Security Modules. This is to combat attacks on packages used by the system. If a package has vulnerability that provides an attacker shell access on the host system, it could lead to malicious actions on the system [20].

SROS is geared towards securing the system in a networked environment. The restriction of nodes and namespaces could be useful, but the system does not implement the basic idea behind this thesis and cannot switch authorized nodes publishing on a topic dynamically. It is still in experimental stage and active development. Some of the functionality and the tutorials are also missing. It is not suitable for the needed access control system as it cannot statisfy the critical requirement 2 of dynamic access control for the multiple controllers commanding the hardware. It does provide the requirements 1, 3, 6, 9 and 10. SROS could be used along the developed system, but the extensive configuration and lack of C++ support may cause issues.

2.2.2 Secure ROS

Secure ROS [21] replaces packages, such as the rosmaster, the C/C++ and Python interfaces for ROS, with its own versions that support secure communications. It does provide the functionality for restricting access on topics to allowed nodes, but the connections are not dynamic and cannot be switched. The secure ROS relies on configuration files to provide the publishers and subscribers which are allowed to connect, which nodes are allowed to get and set ROS parameters and the authorized requesters and providers for ROS services. If no configuration is loaded, the secure ROS system behaves as ROS without any of the enhancements [22].

Secure ROS accomplishes the functionality by securing the connections on IP (Internet Protocol) level and IPsec (IP Security Architecture) ensures the IP addresses are not spoofed [23]. This implementation also means that the connections on the topics, services and parameters are only bound at IP level. The nodes themselves cannot be addressed individually. The nodes are assigned to an IP address and then the IP address may be configured for a topic. However if another node is configured for the same IP address on some other topic, it may still connect to the first topic. The IP addresses can be assigned through aliases, but the documentation is unclear if the alias or the IP is used for the identifier. Most likely it is the IP address the alias points to and therefore the nodes on same machine cannot be identified from each other.

Both the C/C++ and Python client libraries are supported by Secure ROS, but it is only intended for securing connections between different machines on the network. The access control also cannot handle the switching so the system cannot be used for the Care-O-bot 4 implementation. Only the requirement 9 and 10 are fulfilled, while the requirement 1 is only applicable in special cases where each controller and hardware run on different network interface.

3 System design

The upcoming system for safely isolating the controllers from each other needed to be designed before implementation could be started. For the architecture to be designed, the execution method of the main requirement, the access control, of this thesis needed to be determined.

As the research indicated there were no known readymade components which could be used to affect the data used for controlling the robot, a solution needed to be developed. It is known from the provided simulation environment that the data is passed through ROS topics with different messages depending on the interface, for example geometry message Twist containing linear and angular velocities or trajectory message JointTrajectory containing joint angles, velocities and accelerations. It was also known that all internode communications are done through the network layer by default with TCP packets and with UDP packets if requested [24]. The robot runs on Ubuntu computers and therefore the iptables firewall solution can be used to inspect the packets passing the data from one node to another. Using the firewall method to switch between the inputs the existing configuration would not need modifications and the access control could be added afterwards. Only the control signals determining which controller has access to the hardware would need to be added.

Another possible solution thought up was to create switchable multiplexers for each message type. This solution would need individual topics for each controller and the existing configuration needs to be remapped to accommodate the system along the access control components.

3.1 Firewall testing

A simple shell script was written to test if connections from different nodes can be identified, separated and blocked. If this could be done, the isolation of ROS controllers from the hardware interfaces would be done by blocking packets on the network layer and no specialized software is needed to adopt the isolation for different ROS messages.

The "rostopic info" command provides connection information, including the hostname and port number on the nodes publishing and subscribing to the specified topic. The script lists the publishers on the topic and one of these need to be provided as the node the script will block from connecting to the subscribers. The node name is used to filter the network socket information of the local computer for the publisher and the PID (process identifier) of the process are extracted. The script then uses the socket information to get all the ports the process is listening on. These are the source ports for the firewall as the messages originate from the publishers. The PID and port extraction is repeated for the subscribers and the acquired ports are the possible destination ports for the firewall rule. These two port groups are then

matched together so the relevant send and receive parts are grouped and ports with no matching pairs are discarded from the firewall rules. Finally the script reads the process names for the PIDs and provides the user a clear message on the processes and ports to be used for the firewall rules. If the user accepts the provided information, the firewall rule is activated. Once the user wishes to remove the rule, the script will delete it before exiting. The firewall rule is set to reject TCP packets with push flag set so it only affects nodes using the normal ROSTCP connections.

3.1.1 Results

During the experimentation the ports used by topics delivering data had the PSH bit when examined with tcpdump. When a node was not actively publishing messages, the ports had ACK, acknowledgment, set and no data was flowing. These messages turned out to be keepalive [25] messages. If these messages were blocked by the firewall, the connection between nodes was immediately broken and the publisher was observed to try retransmission of the packets. When only the packets with PSH bit set are blocked by the firewall and the node was not publishing any messages to the topic, the connection between nodes was observed to be stable up to six hours at which point the experiment was stopped. After the firewall was deactivated, publishing to the topic from the previously blocked node was not hindered and the subscribers were immediately able see the messages.

The previously mentioned retransmission of packets can be observed immediately when a message was published by the blocked node. The retransmission interval was five seconds for the first four minutes, at which point it was increased to two minutes. After total of 15 minutes the connection attempts stopped and the subscribers were observed to be receiving messages from the blocked node. Inspecting the connections from the nodes revealed that the publisher and subscriber had renegotiated the port used for data transfer. If the firewall was deactivated during the retransmission intervals, the next message passed to the subscribers would occur at the time of next retransmission attempt even if the node was publishing new messages.

These observations disqualify the firewall from being used as the switch between the hardware interface and the controllers. The switchover time between the inputs would be uncertain, but up to two minutes, which is unacceptable for practical applications, thus not fulfilling the requirement 2. The testing and observations were not useless as the methods used were later developed into a firewall node protecting the topics from unwanted publishers fulfilling the requirement 3.

3.2 Final design

The designed system consist of multiple components working together to provide a controllable platform with controller isolation for the hardware. The original request was to implement the system for the Care-O-bot used by the Intelligent Robotics

group at Aalto University. Instead of building a robot specific platform, the system was designed to be used with any robot using ROS with the same messages as the Care-O-bot to control the action of the robot. By not binding to a single platform the system should be usable on multiple robotic platforms, fulfilling the requirement 9. The configuration for each robot is separated into ROS launch files, which bring the system up as configured. The system was designated as ROSGuard as the combined system can protect the robots under ROS from unwanted motions.

It was also acknowledged that the group researches robot co-operation [26] and the system was built to support multiple instances running on different robots. All configuration parameters and control topics used by the system can be configured to be in specified ROS namespace, designated as the robot namespace. Each component accepts a parameter called "robot" and it is used to designate the namespace the control topics and parameters reside in, thus fulfilling the requirement 10. The control topics are created into global namespace using the parameter provided. Therefore the normal ROS naming standards [27] apply to the robot namespace.

The control methods used to control the robot may differ. During experiments the robot may be running a single controller, which is being tested, or the robot is loaded with multiple controllers and a task coordinator, such as SMACH [28]. The access to the hardware resources for each controller need to be activated. To achieve this the system has a central component serving as an interface for the client controllers for the requirement 2. This component is the resource allocator. The controllers themselves can request access from the allocator to the specified resources or the task controller can do it so individual controllers do not need to be modified. In some scenarios and with certain messages the controllers may not need to implement the access reservations as the reservation can be automated by passing the movement messages through a proxy. The proxy can do the access reservation based on the output commands from the controller. The possible configurations are shown in figure 3.1.

The allocator commands the multiplexers responsible of managing the hardware interfaces. As the name implies, the multiplexers have multiple inputs and a single output. The inputs are connected to client controllers and the output is connected to the hardware interface. In the case of Care-O-bot the hardware have multiple interfaces. The actuators can be commanded with multiple message types. For example the arm actuators have five different hardware interfaces. To identify each hardware interface the client controllers can reserve access to the topic or actionserver by their name in the access reservation messages. This however only allocates the one of the possibly multiple hardware interfaces to the client controller. The other interfaces could be used to send conflicting commands to the actuators. This goes against the goal of this thesis as the hardware can no longer be safe.

To rectify this issue each multiplexer node can be assigned to different groups. The groups are used to assign each multiplexer on specific hardware to the relevant group. For example all the five multiplexers on the left Care-O-bot arm are assigned to the arm_left group. Using the group names during the access reservation the whole

hardware resource is assigned to the client controller and the hardware cannot be affected by any other controller. This fulfills the requirement 1. The grouping can also be used to construct metagroups, groups containing multiple actuators. These can be used to construct chains out of the actuators to be reserved in single request. For example on Care-O-bot the left gripper can be included automatically in the arm_left group as the two resources are likely to be used together. The configured system for the Care-O-bot includes this modification and the other groups defined for it are base, gripper_left. arm_right, gripper_right, head and sensorring. Using these identifiers for the access reservation is more intuitive than the topic or actionserver names.

By default the resources can only be reserved if they are not allocated to other controllers and only the controller reserving the resources can relieve them. This fulfills the requirement 7. An override functionality was implemented into the system. Controllers overriding any resource must be ready to accept the robot in arbitrary state. The original controllers are notified by the allocator that their access has been revoked. The previously presented overlapping grouping also provides challenge to the override functionality. It is possible to override a group, which takes partial access from another controller. To prevent any controller from continuing execution with partial resources, the allocator assigns the conflicting group to the overriding controller. This was done to fulfill the requirement 8 as the overrides are intended to be used for resolving faults.

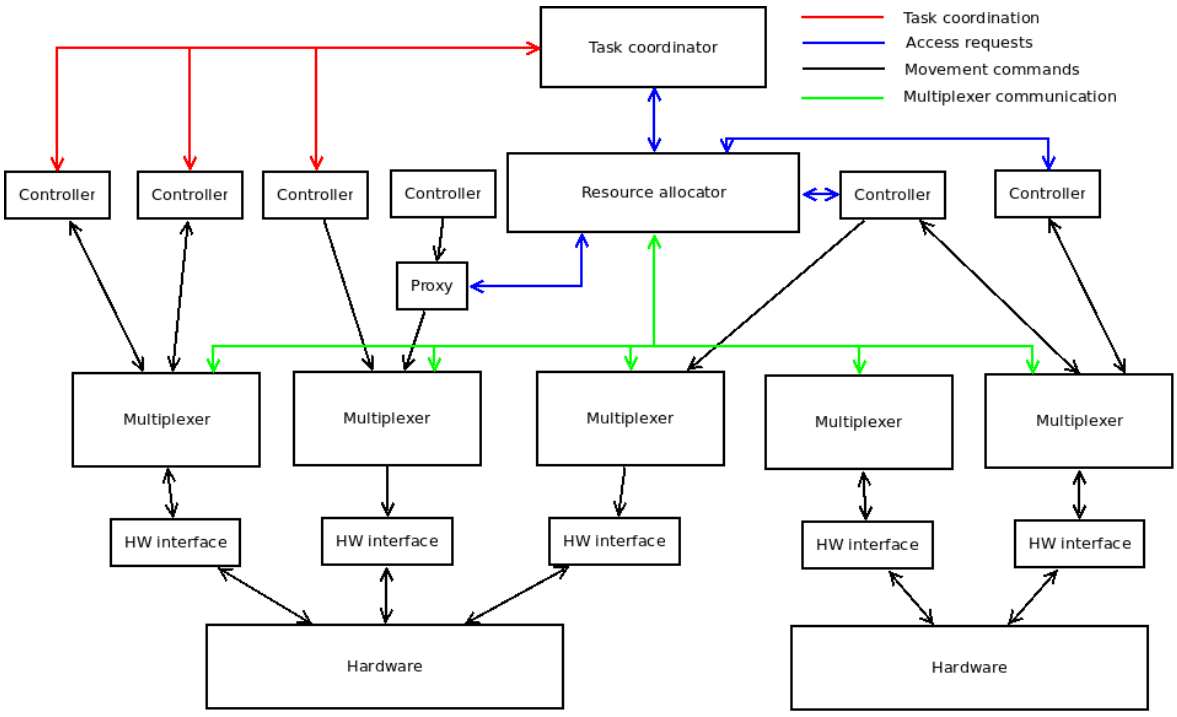

Figure 3.1: *System architecture.*

3.2.1 Additional features

The scope of the system was expanded as the approach using the firewall for control switching was abandoned and the multiplexers were utilized to relay the messages from the controllers. Each multiplexer was tailored for a specific message type. Additional features could be included in the system by modifying the message before passing it to the output. The robotic platform must not harm the humans interacting with it. This can be implemented by restricting the forces the robot is able to exert on the interacting humans to a safe level. The given robotic platform, Care-O-bot 4, does not include any force sensors for controlling the actuators. Instead the system is built to protect the robot and its users from excessive velocities and accelerations, fulfilling the requirement 4. These can be classified as the safety-related speed limits required by the ISO 13482:2014 [8] standard for the personal care robots. Reducing the velocities reduces the risks of injuring other parties in the area the robot operates in [29].

In addition to the configured limits, the system was also built to modify these limits based on the operational state. By the requirement 5 and 6, the robot can be slowed down or even stopped using the system. The stopping of the robot through the multiplexers satisfy the IEC 60204-1 [30] category 2 protective stop if the proper risk assessment and performance levels are met. The same goes for the automatic restart by the control logic. The ROSGuard only provides the framework and the assessment, detection and identification of risks on each platform is left for the end user. The safety related speed control, the slowdown functionality, should be used with caution as the current iteration of the system can meet the requirements only with some of the messages used to move the actuators.

The control over the protective stop states and slowdown functionality were incorporated into a simplified fault message. The cause for the protective stop or the safety-related speed control need to be identified. Unless the states are manually triggered, the source for the mode changes are most likely the fault detection and identification components embodied into the robot control software. The fault system incorporated into the ROSGuard components is not intended to replace any of the components used for fault detection or error recovery, but it is appointed as an easy to use communication channel on the faults and their effects on the system. Usage of the ROSGuard fault message is not mandatory if the protective stop functionality is not needed. The safety related speed limits still apply as they are built in on the multiplexer nodes.

3.3 Components

As defined previously, the ROSGuard system consists of two major components. The resource allocator and the input multiplexers. Other components are the firewall, fault message and the monitoring interface. These three are not vital to the basic operation of the system. They however complete the system by providing services

and security. All of the components are written for ROS using the C/C++ interface roscpp [31].

The resource allocator handles the access requests from the clients. The information about which resources have been allocated is not stored by the allocator. The current status is queried from the multiplexer nodes each time an access request is made. Each robot may only have one allocator at the time and by not storing the information the reliability of the system is increased. The allocator node may crash and respawn or moved between computers without major effects on the operation of the robot.

The multiplexers handle the input switching for each of the hardware interfaces and receive commands from the resource allocator. They also handle the protective stop and speed control. To ensure proper configuration the nodes output debugging information of the inputs and outputs. For example the number of publishers on the hardware interfaces are monitored because the multiplexer nodes should not be bypassed to ensure the controller isolation.

Firewall can be used to lock down the configured topics from future connections. The intended purpose is to ensure the hardware interfaces cannot receive any other messages than the ones relayed by the multiplexer. This could occur as software not adapted for the ROSGuard system begin moving the robot through the original control topics directly connected to the hardware.

The fault message is used to convey the state the multiplexers should operate in. It can also be used by any other component as it can carry information on the events on the robot. The intended purpose is for the client controllers to modify their operation based on the events and possibly stop outputting commands in case the robot is commanded to stop through the multiplexer protective stop.

The monitoring interface can be used to monitor, publish and resolve the faults in the system. It also provides an interface to be used as a software emergency stop for the robot. The multiplexer debug information is also displayed so the users can improve the safety and security of the system by fixing the issues reported by the multiplexers.

In total the ROSGuard system consists of over ten thousand lines of code. This includes the built system and the example controllers. A report was generated with the “sloccount” program and the largest component is the multiplexer consisting nearly half of the codebase. The second largest and the most complex is the firewall, which had multiple issues during the development, but were eventually fixed to provide the needed functionality.

4 ROSGuard messages

All communication between different ROSGuard components, client controllers and the host components is done with ROS topics and their message descriptions are presented in this chapter. It would be logical to use ROS services for the client controllers to request access to the system, but the early testing with the firewall proved the service connections to be unreliable. Using persistent service connections this could be circumvented. The persistent service connections have problems themselves [32] so all communications are done through topics. This also presented the possibility to publish asynchronous messages to the controllers through the access control channel.

All of the messages presented below belong to the rosguard_msgs package and they include a ROS timestamp, which it is omitted in the detailed description. All message types with control commands are enumerated for ease of use and the user does not need to remember numerical codes.

4.1 Access request

The access_request message is used by the client controllers and task coordinators to request control change to the specified resources. The allocator creates the topic the messages are published into and it is shown in table 4.1.

Table 4.1: *access_request.msg*

time	timestamp
uint8	command
	INFO=0 RESERVE=1 FREE=2 OVERRIDE=99
string	target
string	source

The command is the type of the request to run on the system. It has enumerated values of INFO, RESERVE, FREE and OVERRIDE. The default value is INFO when a new message is generated. The command to release resource is FREE instead of RELEASE as the control change on the multiplexers is done to "free" input. There is also no possibility to mistype between release and reserve and as code would result in errors when the enumeration is not found.

The target is the group name or the identifier, topic or actionserver, of the multiplexers commanded through the allocator. The requesting controller needs to know the hardware it wishes to access and relay it in the access request. Unknown targets are rejected by the resource allocator.

The final value is the source, which is the controllers own name or the name of the controller the input is switched to on the multiplexer nodes. The source controller must exist at least on one multiplexer or the resource allocator rejects the message.

4.2 Access response

The access_response message shown in table 4.2 is published by the allocator node as a reply to the access_request message. Every client controller receive the messages and need to filter the information directed at them. These include the responses to the requests and notifications on overrides.

Table 4.2: *access_response.msg*

time	timestamp
uint8	ans
	SUCCESS=0 ERROR=1 RESERVED=2 OVERRIDE=99
string	source
int8	info
	EMPTY=0 SUCCESS_RESERVE=1 SUCCESS_FREE=2 NO_TARGET=3 NO_CONTROLLER=4 CONFLICT=5 TOO_MANY_REQUESTS=6 OVERRIDDEN=99
string	info_text

The source of the request is included from the access_request for the queries and it is also used to inform of the possible overrides on reserved resources. This matches the controller name of the receiving controller.

The target carries the group or the identifier of the multiplexer node which has been the object of the request. On overrides the target declares the resource which has been overridden.

The answer to the request is enumeration of SUCCESS, ERROR or RESERVED for the access control. OVERRIDE is used to tell any controller receiving it to treat the message as a notice on revoked resources. Any response other than SUCCESS, zero, on a request can be interpreted as a failure to the request.

The information fields clarify the received answer. The value can be EMPTY, SUCCESS_RESERVE, SUCCESS_FREE, NO_TARGET, NO_CONTROLLER,

CONFLICT, TOO_MANY_REQUETS or OVERRIDDEN. The OVERRIDDEN is returned for override requests. The success messages are for the relevant request. The EMPTY can only be returned on successful INFO requests. The CONFLICT can be returned by INFO and FREE on unsuccessful operation and by RESERVE in special cases when the multiplexer nodes behave erroneously. NO_TARGET and NO_CONTROLLER are errors from the allocator, which cannot find the source or target of the request. The TOO_MANY_REQUESTS is issued if the maximum of two queued requests per controller are exceeded.

The info_text contains human readable result of the answer to ease debugging of the system. The contained information is more detailed for failed group requests as each conflicting multiplexer name is included in the message.

4.3 FW control

The firewall control message shown in table 4.3 should not be used manually and an UI has been written to control the firewall. All the firewall nodes subscribe to the firewall control topic.

Table 4.3: *fw_control.msg*

time	timestamp
uint8	command
	INFO=0 START=1 STOP=2 ADD=3 REMOVE=4
int8	mode
	PROTECT=0 LOCK=1
string	target

The command to the firewall may be INFO, START, STOP, ADD or REMOVE. The INFO sends a request to the firewall nodes to report the current state on the debug channel. START and STOP enable or disable the firewall operation. ADD and REMOVE are used to load or delete topics on the firewall nodes.

The mode is used when adding topics to determine the operation on the requested topic. The options are PROTECT and LOCK, which correspond to the firewall operation modes. These are explained in the chapter 8 for the firewall.

The target is the fully resolved ROS topic to be added or removed on the firewall nodes.

4.4 FW debug

The firewall debug message in table 4.4 contain the status updates from the firewall nodes or responses to the information request to a control message. Each firewall node publishes these messages and the firewall controller subscribes on the debug topic.

Table 4.4: *fw_ debg.msg*

time	timestamp
int8	status
	STOPPED=0 RUNNING=1 ERROR=-1
string	machine
string[]	topics

The status of the firewall is either STOPPED or RUNNING and for error messages the status is marked ERROR. The machine the message came from is included in each message to identify the source of the message.

On normal operation the topics each firewall has are returned along the debug messages, but on error messages the topics array is filled with human readable information on the error.

4.5 MUX control

The multiplexer control message presented in table 4.5 is published by the allocator node to the mux control topic and all multiplexers on the robot subscribe to it for receiving commands.

Table 4.5: *mux_ control.msg*

time	timestamp
uint8	command
	INFO=0 ACTIVATE=1 FREE=2 CREATE=3 DELETE=4 OVERRIDE=99
string	source
string	target

The commands are INFO, ACTIVATE, FREE, CREATE, DELETE and OVERRIDE. The info commands query the multiplexers if the source of the request is authorized to execute changes for the control switch. ACTIVATE and FREE are used to reserve and release the resource for the source of the request. OVERRIDE commands command the multiplexer to the override mode, which is explained in the allocator and multiplexer chapters. CREATE and DELETE were added for dynamic input creation, but the current system does not support them.

The source is the controller name commanding the multiplexer and the target is either a mux group name or the individual node identifier the command is to be executed on.

4.6 MUX debug

The response to the multiplexer command is given on the debug channel with the message shown in table 4.6, which the allocator subscribes in for the responses. The debug message is also used to publish any events on the multiplexers and the user interface for monitoring the system also subscribes to the topic.

Table 4.6: *mux_ debug.msg*

time	timestamp
string	topicname
string[]	groups
string	input
string[]	available_inputs
uint8	mode
	NORMAL=0 SLOW=1 SLOW_STOP=2 FAST_STOP=3
int8	info
	SUCCESS=0 FREE=1 MODE_CHANGE=3 CREATED=4 DELETED=5 RESERVED=6 CONFLICT=7 UNAUTHORIZED=8 LIMITER=9 INPUT_ON_INACTIVE=10 OVERRIDE=99
string	target
string	source

Each message includes the identifier of the multiplexer as a topic or actionserver name and all the groups assigned on the multiplexer are also relayed on the message. For status messages the current configured input controller is relayed back. Upon unauthorized access to the multiplexer, the violating controller is returned on the input field. The same is done for messages warning of active communications on an inactive multiplexer input and of excessive velocities detected.

The current operation mode of the multiplexer, NORMAL, SLOW, SLOW_STOP or FAST_STOP, is relayed along with the information on the cause of the message. SUCCESS, FREE and OVERRIDE for successful control changes and RESERVED, CONFLICT and UNAUTHORIZED for declined requests. The INPUT_ON_-INACTIVE is used to inform of unintended input as explained before and LIMITER is used to signify that the current input has exceeded velocity or acceleration limits configured on the multiplexer. CREATED and DELETED are unused as the current implementation does not support dynamically adding or removing input controllers.

4.7 MUX limit configure

As the design calles for the ability to safeguard the actuators from excessive velocities and actuators, the feature needed to be configurable at runtime. Restarting the system each time to test the limits of motion would lead to wasted time. The multiplexer limits can be individually reconfigured on single control topic the mux nodes communicate on by using the configuration message presented in table 4.7.

Table 4.7: *mux_limit_configure.msg*

time	timestamp
string	target
uint8	mask
	N_LIN_A=0 N_LIN_V=1 N_ANG_A=2 N_ANG_V=3 TIMEOUT=4 S_VEL_RATIO=5 S_ACC_RATIO=6
float64	n_linear_acc
float64	n_linear_vel
float64	n_angular_acc
float64	n_angular_vel
float64	timeout
float64	s_vel_ratio
float64	s_acc_ratio

Each message needs to define the target by using the identifier of the multiplexer node. Mux groups cannot be used as identifiers. The component of the limit to be modified is set by bytemasking an 8-bit value. Only the values masked will be changed on the multiplexer. The mask values for normal linear and angular velocity and acceleration are N_LIN_V, N_LIN_A, N_ANG_V and N_ANG_A. The timeout value on can be changed with the TIMEOUT mask. The slow operation mode is configured as modifier to the normal velocities with S_VEL_RATIO and S_ACC_RATIO. The velocity ratio can be value of 0 to 1, but the acceleration is locked between 0.3 and 1 as setting zero accelerations would lead to inability to change the current velocities. A special mode was added by sending a message with zero mask and zero timestamp. The queried multiplexer will reply with the current values on the same topic.

4.8 Fault

The fault message is used to convey problems in the robot for the ROSGuard system. A fault is defined as a hypothesized error or a failure on the robot or the environment [33]. The multiplexers needed a message to change the operation modes based on the events on the robot. There was no existing generic fault message the nodes could use. The ROS diagnostics messages carried some information on the errors on the robot, but it was unsuited for the ROSGuard system due to inflexibility. The fault message was made to be used as a simplified failure notification for the system. The categories presented here are the initial generalization for a service robot. The focus on this thesis was not the detection and identification of failures and no completed listing of faults were available. Instead surveys of faults were looked up and simplified [13, 34, 35]. The resulting message is shown in table 4.8. More of the usage of the faults is presented in the following chapters.

Table 4.8: *fault.msg*

time	timestamp
string	source
string	id
int16	type
	DEBUG=0, ACCELERATION=1, ENVIRONMENT=2 CHASSIS=3, CONTROL=4, NAVIGATION=5 E_STOP=99, RESOLVED=-1
uint16	description
(acceleration)	LOW_VIBRATION=1, HIGH_VIBRATION=2 THRESHOLD_EXCEEDED=3, ROBOT_TILTED=4
(environment)	COLLISION=1, PINCH=2, HAZARDOUS_FOR_ROBOT=3 HAZARDOUS_FOR_HUMANS=4
(chassis)	POWER_WARN=1, POWER_CRIT=2, STUCK=3 NO_CONTROL=4 OUT_OF_CONTROL=5
(control)	CRASH=1, SENSOR_NULL=2 COMMUNICATION_ERROR=3
(navigation)	NO_PATH=1, LOST=2, CONFINED=3
(e_stop)	PRESSED=1, ACTIVE=2
uint8	severity
	UNKNOWN=0 ENVIRONMENT_WARNING=1, ROBOT_WARNING=2 HUMAN_WARNING=3, ENVIRONMENT_HARM=4 ROBOT_HARM=5, HUMAN_HARM=6
int8	action
	NORMAL=0, SLOW=1, SLOW_STOP=2, FAST_STOP=3
uint8[]	data
string	human_readable

The message includes the source of the fault. It is used to identify the component publishing the fault as the system is built to support multiple monitoring components to identify failures and hazards in the system and environment. Each fault is unique and an identifier is attached to each message. New messages with same identifier are ignored by the multiplexers unless the fault is resolved or the operation mode of the robot is downgraded from the previous ones. The identifier is not allowed to be less than three characters or the system will ignore the message.

The fault types are reduced to ACCELERATION, ENVIRONMENT, CHASSIS, CONTROL, NAVIGATION, E_STOP and the default DEBUG. The type is also used to signify if the fault has been handled and the fault can be removed by setting it to RESOLVED. Further negative values are reserved to set the pending fault timeout described in the next chapter. The fault types are further refined with descriptions. The default zero is defined as UNKNOWN. The descriptions for acceleration are LOW_VIBRATION, HIGH_VIBRATION, THRESHOLD_EX-

CEEDED and ROBOT_TILTED. For environment they are COLLISION, PINCH, HAZARDOUS_FOR_ROBOT and HAZARDOUS_FOR_HUMAN. Chassis descriptions are POWER_WARN, POWER_CRIT, STUCK, NO_CONTROL and OUT_OF_CONTROL. The software control is reduced to CRASH, SENSOR_NULL and COMMUNICATION_ERROR. Navigation has NO_PATH and E-stop has PRESSED and ACTIVE.

Each fault also has a severity with values of ENVIRONMENT_WARNING, ROBOT_WARNING, HUMAN_WARNING and the same for HARM. The information intended for the multiplexer nodes is set in the action, which may be the default NORMAL, SLOW, SLOW_STOP or FAST_STOP. A free data field is provided for applications which need to pass more data in the fault message. Finally a human readable text is attached to the fault for easy inspection.

5 Faults

The system was designed to have more features than the basic input switching in the later stages of development by the request of the research group. One of these features was the ability to manipulate the input depending on the state of the robot, the requirements 5 and 6. A fault subsystem was also requested for the implemented environment and these two features were combined into single fault message and robot specific topic. Without the input manipulation the fault message would be simple information from the nodes detecting and identifying the faults. By adding the command for multiplexer states, the detecting nodes can determine if the failure requires the robot to slow down or stop movement. The slowdown functionality is for safety-related speed control and stop states are for category 2 stop as the system can recover from them automatically. It is not a true emergency stop as the ROSGuard system only impedes movement with software restrictions and the emergency stop is prohibited from automatic resets [8]. The system can be interpreted as the protective stop function required by ISO 10218-2:2011, which is for industrial robots but is still a good guideline for any robotic applications. As the ROSGuard system is designed to be a generic platform and only provides the framework, the end user needs to ensure the robot platform meets the performance levels [36] defined for the operation. For example the ISO 13482:2014 [8] for personal care robots.

The fault system can be bypassed by the end user if so desired. The users are not obligated to use the faults if the different operation modes for multiplexers are not needed and possibly some other system is used to monitor the robot. This increases the flexibility of the system as better suited components may be used for coordination on the events on the robot.

One of such systems is presented by Crestani et al. for mobile robots [37]. However the paper does not provide a practical example, but as a whole the ROSGuard system provides the needed components for the implementation for most of the proposed functionality. Protective stop, dedicated fault detection and resolving controller nodes, which can override the normal controllers to achieve the recovery, are supported by ROSGuard components. The fault messages tie the system together as it can be used to inform the different components, which can then form a decision on the actions.

5.1 Architecture

Certain rules must be established as a single topic is used for all the communication used by the fault system. The fault message carries information on faults and if these faults have been resolved. A way for the components subscribing to the fault message to differentiate between these two is the type of fault. The enumerations are arranged so any positive value can be interpreted as a new fault, zero is for debug and the negative values are for resolved faults or faults which are not yet active.

Any component may subscribe or publish into the fault topic. Publishers need to track the published messages and ensure they do not flood the topic with repeat messages. One message per fault is allowed and it is tracked with the unique identifier. The identifier has no restrictions other than it should be more than two characters long. A reasonable identifier would be a short description of the fault, but any random or hashed identifier is acceptable. To enforce the one message per fault rule, the subscribers can keep track of the received faults and ignore further messages with same identifiers. The multiplexer nodes keep track of the faults, but allow replacement of the message in storage if the message degrades the operation mode of the multiplexer. This is to ensure the faults slowing or stopping the robot get through in cases where the fault publishers escalate the issue using the same fault identifier.

The division of publisher and subscribers is left to the end user, but the intended operation of the fault publishing and resolving of the faults requires two way communication on the topic. The basic components are the fault detection nodes, the fault processing nodes and combined controllers. Incorporating all fault detection and identification features on single node or computer can prove problematic as the processing power on mobile robot may be limited. Distributing them on multiple nodes, computers on the robot or external should help the development and the nodes can specialize in specific scenarios if needed.

5.1.1 Fault publishers

The nodes publishing the faults may be conducting fault detection and identification in the traditional sense by analyzing the ongoing failures. But as the system can slow the movements and even stop if necessary, the faults are also protecting the robot, environment or interacting parties from future harm. The safety-related speed control and protective stop features fall under these nodes. These may be for example a slowdown of moving components if personnel are detected in the safeguarded area of the robot and automatic stop if they are detected in the protected space [8]. Stabilizing act may be for example if the robotic platform is unbalanced or on uneven platform and the sensors detect the vibration or tilting of the body, the detecting component can decide to command the multiplexers to slow down and try keep the platform from losing balance. Another example where the robot may need to be stopped is a typical collision where the extremities or the base of the robot hit an unknown object and the maximum allowed acceleration thresholds are exceeded. Even faults without effect on the multiplexer nodes are useful as ongoing faults indicating problems in the actuators or critical power levels can be used to modify the behaviour of the system.

Publishing a fault is done by creating the message, setting the current timestamp and the name of node publishing the fault. Next the fault type, description, severity and action are set depending on the fault. The controller may append extra data on the fault message if it is needed for the nodes receiving the message. A clear human

readable description of the fault needs to be added for the users to have an easy understandable fault description. The identifier of the fault needs to be over two characters long and preferably unique.

5.1.2 Fault resolvers

The fault resolvers are components dedicated for reacting to the events in the system. They are responsible for clearing the ongoing faults in the system. Usage is freeform as the system only provides the messages. For example in cases where a low power warning has been published, a resolver may wait and try reserving the resources to lead the robot to a charging station. If there is no central coordinator, this may fail and the controller does not activate. Once the power levels reach critical stage the controller may start forcing the access so the robot does not suffer from power failure. Of course forcing the access may not be safe depending on the ongoing task so either the user needs to decide if this is allowed or the fault resolver may have knowledge of the executing task and decide it automatically. The ROSGuard access reservation component can provide the name of the controllers currently accessing the needed resources and this information is returned on failed access reservations.

Resolving a fault can be accomplished by taking the identifier of the resolved fault and setting it as the identifier of the new message. The name of the node should be added as source of the message as it is responsible of resolving the fault. The type needs to be set as RESOLVED and the timestamp to the current time before publishing the fault. This same can be done by taking the old fault message and modifying the type, timestamp and source if the resolver node saves the fault messages. The other values in the RESOLVED messages do not affect the system.

5.1.3 Combined controllers

A combined controller is as the name implies a combined fault detector and resolver in the same package. In its simplest form it may be the safety-related speed controller, commanding the platform to slow down or stop in the presence of human and then clearing the fault once the safe distances are re-established. The presented example controller does not command any movements.

More complex and specialized controllers may be developed for the situations needed. An edge case, which can be handled by a dedicated controller tailored for the task. The example provided in the resource allocation chapter 7.2.3 on overriding resources could be a combined controller as it is used to recover from unexpected collisions during navigation.

5.2 Pending faults

The system also provides a way to publish delayed faults. These are intended to be used when the task being executed is known to cause issues if it not completed in the predefined timeframe. The controller executing the task publishes the fault as it executes the relevant event leading to the possible hazard. Once the hazard has been passed the pending fault can cleared like normal ones by sending the RESOLVED type with the appropriate identification.

If the pending fault has not been resolved before the set time, it is published as a normal fault. This is done by the ROSGuard HTTP user interface monitor and it is needed for the pending faults to work. As the delayed faults are communicated on the same topic and message as normal faults, the type of the fault is set to negative value. This ensures that other controllers ignore the messages as real faults can only have positive values.

The pending faults can only use the first 99 descriptions of faults and the type is allowed for positive values within reason as the messages are transported as 16-bit unsigned integer. The amount of time the faults are pending is encoded in the type of the fault as negative value. It is calculated as by taking the negative value of the minutes the fault times out to be published and subtracting one from it.

The minimum delay is two minutes as the value of -1 is reserved for the RESOLVED enumeration. The accuracy the messages are published on timeout is $\pm$ 15 seconds at the default HTTP monitor update rate. The time is transported as 16-bit integer so the maximum delay is almost 23 days. The real type of the fault is encoded to the description by multiplying it with 100 and adding it to the description, the monitoring node decodes these back into normal faults and stores them for publishing.

An example of a pending fault could be a request for the robot assistant to issue a warning if the owner has not taken medication within thirty minutes from a specified location. In this case the system is used for crude timing tasks, but publishing a fault at the time of request keeps track if the medication has not been served. If the fault has been published it can alter the behaviour of the robot and a monitoring node may forward the specific fault to a supervising party. The example can be further developed by the supervising party overtaking the robot into teleoperation for observing and helping the owner, possibly avoiding the need to travel to the owners residence.

5.3 ROS Diagnostics to fault

The ROS diagnostics layer was rejected as the activation method because the multiplexer operation modes would have conflicted with the existing diagnostics messages. The diagnostics messages provide information on the state of the robot and these may be wished to be imported into the fault messages so controllers may use the state to determine their actions.

The diagnostics to fault package provides a node, which can translate the diagnostics messages to the rosguard_msgs/fault message. As the diagnostic is mainly information on the robot state, the fault type is defined as CHASSIS for all messages published by the node. The diagnostics message is usually not enough to determine the detailed description on the fault message and it is defined as UNKNOWN. The same value is defined for the severity of the fault.

The node translates all diagnostics messages with elevated diagnostics status, ERROR or WARN, into faults. By default they do not affect the multiplexer operation, but a simple name matching has been provided to filter the messages. Three different filters provide varying responses to the diagnostics messages. The filters accept an array of strings as parameters and they are matched against the name of the diagnostics message. If a diagnostics name contains the defined string, the filter is activated. The action_normal translates messages with diagnostics status of WARN into SLOW state for the multiplexers and ERROR into SLOW_STOP. The actionfilter_error_stop does not modify WARN diagnostics and only translates ERROR into SLOW_STOP. The final actionfilter_error_slow behaves similarly, but instead converts the ERROR to SLOW, not stopping the robot.

5.3.1 Care-O-bot 4 power state monitor

Although the provided environment for COB supplies ROS diagnostics on information for the actuators and sensors, it does not include the power and battery status. The robot is publishing PowerState message from the cob_messages package informing the voltage, current, remaining runtime and other statistics. The robot also has node which can be configured to actuate light or activate speech synthesis to inform the user if the battery levels are low.

As the information is not in the diagnostics, a package was created to include Care-O-bot chassis specific fault monitors. The rosguard_cob4_chassis_fault_monitors currently includes a power_state_monitor node, which can be configured to publish the ROSGuard fault messages. The low and critical battery levels are configured as percentages. The node will publish CHASSIS fault with POWER_WARN and POWER_CRIT if the battery levels drop below the configured values and the robot is not charging. The node keeps track of the published faults and it can be configured to automatically clear the faults when the robot is connected to the charger.

5.4 Future improvements

The fault message presented here is a working placeholder, which is to be expanded as the need arises. The target was to deliver an easy to understand and configure fault information for the applications and end users. Using this information may be difficult as the receiving end needs to manually filter the desired information by subscribing to the topic and filtering the unwanted messages. In future this can be

improved by providing a library keeping track of the faults and filtering only the desired faults for inspection by the client component.

The usage of the fault system is not mandatory for the controllers. The usage is flexible and may even be completely ignored if the multiplexer nodes do not need to be commanded. Once different approaches on the functionality are made, they should be documented for reference on how the system can be utilized.

The usage of different controllers for resolving the faults needs work as there may be multiple possible ways to resolve an issue. The coordination is undefined and overlapping components may try resolving the issue simultaneously and the controller may resolve the issue by trying to force access for themselves, resulting in endless loop of access reservations. Ideally this should not happen as each fault would have its own response on the system or the system has dedicated analyzer deciding the course of action on faults.

The pending faults have poor resolution and excessive maximum delay for normal operation. It is not intended for precise operation as it is a fallback for tasks failing. If more resolution and accuracy is needed the calculations and update rate in the monitoring node publishing the faults can be modified.

6 Input multiplexer

As the initial testing proved the use of single topic for multiple controllable inputs to be unacceptable with the firewall due to long control switch times, the inputs from each controller were separated from the hardware target. By designating an input interface for each defined controller, separate from the hardware, some degree of isolation is already achieved. Also additional features were be added to the system as the messages needed to be relayed to the hardware. The need to relay the commands from the selected controller to hardware interface is the main feature of the multiplexer, or mux for short. The use of firewall for input multiplexing would have given the option to use it on any topic, but the requirement of creating a subscriber for each controller input leads to a need to specialize each input to match the output to the hardware interface. Fortunately the approach for robot hardware control interfaces is widely standardized on ROS to handful of message types. The use of these messages and action types allows for additional features including velocity and acceleration limits along the possibility of stopping or slowing down the output to the target interface for the system requirements 4 through 6.

These limits are configured in the robot specific launch files, but an interface for reconfiguring these at runtime has been provided incase an experiment needs to be performed at varying limits. Reconfiguring is done at a single topic, which can also be used to query the current limits. The set limits are used for the normal operation mode. Slower operation mode is achieved by scaling these parameters. Other mux specific topics include the mux debug and the control switch interface. Each mux node receives the control change command from the robot specific allocator as detailed in section 3.2. The slowdown and stop feature activation were integrated into the ROSGuard fault messages and each mux node keeps track of the ongoing faults affecting the multiplexer operational states.

Each of the mux nodes are individually identified by the output topic or action name. The mux nodes can be individually addressed by these in the access control requests. For better control over the hardware each node can be assigned multiple names, designated as groups, that can be called over the access reservation. A hardware resource can have multiple driving interfaces with different control methods. Grouping these into under single name can improve immunity against accidental motion. When using the group name to reserve the resource, all the interfaces associated with the single hardware resource are diverted to the controller with access reservation. Using these groups even multiple hardware interfaces can be packed together as an unit. For example the arm actuators and attached grippers can be organized as a whole arm or tree structures can be build to ensure the whole chain from robot base to the designated actuator is under single controller. A special group called “robot” is embedded into each mux node to be used as a global group, which can be used to reserve the whole robot.

When using tree structures or the global group, the reserving controller does not need to actuate all the resources. The mux nodes do not check if the reserved controller

has an input configured. In these cases the input switch is done and the output is locked to null output. This is one of the reasons the OVERRIDE in the ROSGuard allocator can take over overlapping groups without issues.

6.1 Faults in multiplexer

The simplified fault system was integrated into the multiplexer nodes for the function of software emergency stop and speed control. Separated from the hardware emergency stop, the soft E-stop can be used for automated stopping and slowing down of the robot platform with possibility to lift the soft E-stop within software for the platform to operate normally without constant user intervention.

The mux nodes implement the stopping and slowing down functions of by modifying the received messages from the controllers before passing them to the hardware interface. The four operations modes match the action presented in the ROSGuard fault message: NORMAL, SLOW, SLOW_STOP and FAST_STOP. In normal mode the outputs are passed through to the hardware unless any of the configured velocity or acceleration limits are exceeded. If the limits are exceeded, the multiplexers clamps the outputs to the maximum allowed values and publishes a debug message informing about the exceeded velocities. The slow mode reduces the limits to the configured ratios, slowing down the incoming and ongoing movements, to the maximum allowed levels if the interface supports it. The slow stop is first of the modes stopping all movement. It allows the movement to decelerate to zero if the interface supports it. In the fast stop mode the outputs are forced to zero instantly.

For these modes to operate they have a priority based on the faults received. The fast stop has the highest priority followed by slow stop and slow. The normal operating mode has lowest priority. This is done by tracking the incoming faults and their identifiers. Each fault affecting the state of the mux is saved into memory. The faults can be removed from the mux node by resolving the fault. All the faults leading to the operation modes need to be resolved before the operation mode is elevated. Both of the stopping faults need to be resolved before the mux allows movement and all the slowing faults must be resolved before the normal operation mode is applied.

The exception to this rule is the OVERRIDE coming from the allocator. The override is meant to give access to a controller for it to recover from a fault. To achieve this the robot needs to move. Therefore the OVERRIDE can lift the multiplexers from stopped state into slow state even with active faults, partially fulfilling the requirement 8. A new stopping fault received by the mux node will bring the movement to halt, so the override mode is not a complete control over the robot movement. The override needs to be activated again for the robot to move in these conditions. Freeing the resource cancels the override operation and the priority of operation modes is again enforced. If override is issued on a moving actuator, the multiplexer will force the current actions to be halted and the overriding controller will not receive a resource with prior commands.

6.2 Configuration

All of the multiplexer nodes share the same configuration parameter names in different interface types. The ROSGuard namespace is passed with the robot parameter as with all other components.

The output to the hardware interface is configured with the output_topic parameter. For single topic interfaces this is the topic and for action interfaces the parent name before the goal of the action is used. The inputs for the controllers are formed after the configured output with the naming scheme of "$output_topic"/mux_-input/"$controller_name". In the case of single topic interface, such as geometry_-msgs/Twist, the controller name is the input and for action interface the action server can be pointed to this as the action control topics are generated after the name. The input controllers are defined in the input_controllers parameter and it is a space separated list of the controller names. Note that these names are passed to the allocator node for checking the incoming requests so they need to match on the client controllers and ROS naming standards [27].

The group names are configured on the groups parameter which is space separated and like the controller names. These are passed to the allocator for input verification. The global group "robot" does not need to be configured manually. The two timing parameters, updaterate and timeout, are interface specific and the defaults are 10Hz for the updaterate and 3 seconds for the timeout.

The input limits are defined for linear and angular parameters. The linear velocity limit is defined with lin_vel_limit and acceleration with lin_acc_limit. For angular motion these are ang_vel_limit and ang_acc_limit. These limits cannot be negative and absolute value of the parameter is used incase a negative value is defined. Slowing down the movement in slow mode is achieved by scaling the acceleration and velocity limits with the slowdown_a_ratio and slowdown_v_ratio. These parameters are checked for sane limits. The velocity ratio is to be between 1 and 0 so the system cannot be sped up in the slow mode. The acceleration limits are to be between 0.3 and 1. The minimum needs to be set as with acceleration scaled to zero, the system cannot change any velocities including the ongoing motions. Default ratio for the velocities is 0.3 and 0.9 for accelerations. For some interfaces the parameter jointStateFeedback needs to be configured to point to the jointState message of the actuator in question. Without this the limit calculations do not work and the messages are passed through without checking the limits.

6.3 Reconfiguring input limits

The possible need for reconfiguring specified mux nodes during experiments was presented before. This functionality is accessible through the mux_control/mux_-limits topic under the ROSGuard namespace. Each mux node can be reconfigured by sending rosguad_msgs/mux_limit_configure message to the topic. The target

is specified by the unique name of the mux node, the output topic name. Group names are not allowed as a target. Each of the configuration parameters presented before for the input limits and slowdown functionality can be modified. To configure a parameter, the values need to be sent along with a command to load the defined values. This command is a simple bytemask for each parameter, 1 for loading the value and 0 for ignoring the received value and using the old one. The bytemasks are enumerated. For example in C to enable the sent linear velocity limit the bytemask is set by bitshifting the enumeration.

```
mask = (1<<(rosguard_msgs::mux_limit_configure::N_LIN_V));
```

The current configuration can be queried from the same topic by sending a message with zero timestamp and zero mask to the target. This can be used to read the current values and increment some value before sending it back. This may be used for example in cases where the current limits are restricting an experiment and they needs to be increased. Caution should be used when increasing the limits to ensure safety for the robot, user and environment.

6.4 Error and debug reporting

The limits specified previously are enforced by the multiplexer nodes. Exceeding these limits leads to error reporting on the mux_control/debug topic under the ROSGuard namespace. The offending controller name, mux identifier and operation mode are supplied in the debug message. This message is common when the movements are slowed down in the slow mode and the mux nodes need to limit the output. Under normal operation this tells that the current controller is commanding the robot with excessive motions and should be reconfigured to stay within the safe limits.

Other areas the debug information is sent include the checks on number of publishers on the output topic. If there are multiple publishers on the target interface the purpose of the multiplexer could have been bypassed as a node is directly publishing into the hardware. The same checks are performed on the input controllers as they are designed to only have single publisher per input, but this information is only printed as an error to the console as accidentally having two controllers on input topic is more unlikely to happen than controllers on the original control topic.

The subscriber per input was designed for separation so only one can access the hardware interface at a time. The inputs are monitored for activity and if any is detected while an input is not the one currently active, it is reported on the debug message as input on inactive error. This is to help debugging the system as the controllers are not supposed to output any commands while they do not have the resource. This can indicate of a misconfigured or a faulty controller. Operation mode changes due to faults are also reported in the debug messages along the activation of override modes.

The mux debug is also the communication back to the allocator so it can monitor

the success of reservations and query the status of the mux nodes. Through this channel the input controller names and groups are passed to the allocator during the autodiscovery of multiplexer nodes.

6.5 Interfaces

As the robot movement is usually executed by moving joints or wheels, the number of interfaces used to command these are limited. The movement of the robot on non-walking types is commonly not a low level motor driver, but a system where the robot base can be commanded to move and rotate in different directions depending on the drive system. The ROS message geometry_msgs/Twist is the input for these and the navigation stacks included in ROS output this message type.

For joint articulated actuators there are multiple messages and actions. A common action is the control_msgs/FollowJointTrajectoryAction, which can be used for example with MoveIt! Movement Planning Framework. Additionally all the interfaces used by Care-O-bot 4 are implemented. These include the trajectory_msgs/JointTrajectory, Twist and std_msgs/Float64MultiArray for joint positions and velocities.

The interfaces can be divided into two categories based on the operation principle. Continuous output and event based. The event based interfaces have a single message containing multiple movements with defined timeframes. The FollowJointTrajectory-Action and JointTrajectory belong into this category as they send trajectory points with goaltimes to the actuators. In the continuous output interfaces the messages are passed periodically for the actuators to move. These include the Twist and Float64MultiArray interfaces. The continuous type are required to stop outputting after publishing zero movement messages for two times the timeout if there are no input messages. This is to ensure that the hardware resources do not receive conflicting commands on the different interfaces.

6.5.1 Twist

The Twist message in the geometry_msgs package is a six axis linear and angular message. The multiplexer treats this as a continuous velocity message type as it used for example to move and rotate a robot base and end actuators as long as the messages keep flowing.

The continuous operation means that the mux will publish to the output at the update rate regardless of the input frequency. The timeout is used to zero and shutdown the output when the input messages stop. For example if the input is commanded to 1 unit on linear X and the output has reached the desired value and the input messages stop streaming, the output is held for the timeout and after this the desired output is set to zero. The output is decelerated within the limits. The output is switched off after output messages with zero values have been published for twice the timeout as defined earlier.

The twist messages work around zero output so they do not have any feedback from the actuators. Zero as output is considered a stopped motion so the fast stop mode publishes zeroes when activated. Acceleration control works in the twist interface so slow stop uses normal acceleration limits to bring movement to halt. The slow mode uses the defined acceleration ratio to bring the velocities to the new levels.

The twist interface publishes a zero output command as a last message on shutdown to ensure the actuator stops movement even if the node is shutdown while outputting data.

6.5.2 Float64MultiArray joint velocities

The Float64MultiArray works in similar way to the twist node that it is outputting continuous stream of data containing angular velocity information. It does not require feedback from the actuators, but to work it needs to know the number of joints it is operating on. This information is extracted from the input messages and saved. As the number of joints is a variable, the node does not publish a zero output message upon shutdown. This should not prove a problem as in Care-O-bot 4 simulation the testing showed the default timeout on the hardware interface to be under half a second and the zero on shutdown is a corner case.

6.5.3 Float64MultiArray joint positions

The joint position version of the Float64MultiArray requires a feedback from the actuator jointState in order for the node to know the current position of the joints. Angular position information is received from the input and it is bound to the velocity limits approximately regarding the update rate, start and end positions.

In simulation using the joint position interface resulted in the actuators using full acceleration in the movements. As only the position information is passed, the acceleration limits cannot be set. Using higher update rate the movements can be divided into smaller steps, but even at 50Hz the movement was observed to be jerky in simulation. Still even with the default 10Hz rate the actuator in simulation moved at much slower pace than directly relaying the end position to the hardware interface. This interface does not command the current position back to the actuator at shutdown as it could prove hazardous if the jointState is delayed, corrupted or misconfigured. For the same reason it does not send the current position to the actuator at control changes to stop the movement.

6.5.4 FollowJointTrajectoryAction

The first event based message the multiplexer was developed on was the one used by the cob_dashboard for commanding the simple demonstration movements on the Care-O-bot 4. The simulation environment was heavily used to test and debug the

multiplexer as the message format is far more complex than the ones used by the previously described continuous interfaces.

The mux does not use the updaterate or timeout parameters as they have no purpose on the single message based execution. The mux reacts to each incoming message and modifies the contents to match the desired velocity and acceleration limits. The incoming messages give the joint angles on the path and the goaltimes between each point. The starting point is not given, but the actionserver on the hardware interface provides the current joint angles and velocities for the mux to calculate changes to the first point on the path. The whole action interface is replicated for each input controller and the ROS actionclients can use mux_input/controller_name as the target for constructing the actionclient. The feedback from the hardware interface is distributed for each input, but only the selected input can send commands to the hardware.

As the mux handles the velocity and acceleration limits on each message passed to the output action, the slowdown functionality is not available. A slowdown functionality was developed for single endpoint trajectories by extending the required goaltime to the point. The ROS Trajectory Controller does trajectory replacement [38] when a new goal is inserted on ongoing movement. This experiment resulted in two problems. First the trajectory replacement uses the ongoing velocities to apply smooth accelerations over the whole goaltime. This resulted in the joint angles overshooting the target on fast movements as it decelerates slowly before changing direction to return to the target point. This is shown in figure 6.1 where the fast trajectory with single endpoint is replaced in the middle of movement with the same trajectory, but with a longer goaltime.

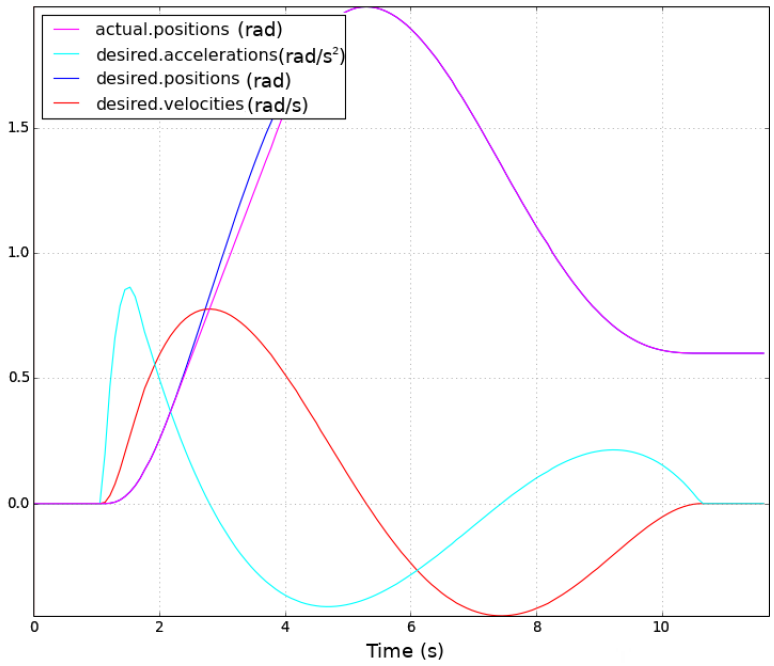

Figure 6.1: *Overshoot on joint caused by trajectory replacement.*

The problem was fixed by generating an intermediate point in front of the current joint angles with the new velocity limit as the velocity goal in the message, if current velocity exceeded the new slower limit. The point can be calculated from the given angular acceleration limits and the velocity difference. The trajectory controller will meet this intermediate point and then proceed to the endpoint with the extended goaltime, slowing down the actuator.

This solution worked only for the single endpoint trajectory where normally trajectories include multiple points to constrain the movement to a planned path. For the slowdown function to work, these points would need to be tracked and the actionserver only reports completion on the whole trajectory. The second problem is the actionserver only accepting unique trajectory names as goal identifiers. This was fixed by generating a new identifier for the replacement trajectories, but mapping them to the original identifiers for the actionclient on the input controller. In the end the slowing down functionality was not reliable and the trajectory replacement and goal identification remapping were removed.

The mux node does not slow down the ongoing motions. New trajectories after the slow mode has been entered will receive the new limits. The mux does not support slow stop and all motion is stopped as in the fast stop. All goals are cancelled which leads to the hardware interface to stop motion. Later in testing this was found out to cause hardware faults when the arm actuators were stopped from motion. The mux also stops all motions between control changes and during the shutdown.

6.5.5 JointTrajectory

The JointTrajectory works almost identically with the FollowJointTajectoryAction as the message is the same as inside the action goal. The output_topic is defined as the command interface, but the JointTrajectory also includes a state topic, which is automatically included for the client controllers. The command input and state can be found under mux_input/controller_name/command and state.

The JointTrajectory interface does not include any provisions for slowing down ongoing movement or slow stop. Only the cancellation of movement is supported and it is done on control change and on shutdown.

6.6 Care-O-bot 4-8 usage

The robot has distinctive groups based on the hardware. The Schunk arms and the head joint use all of the interfaces presented above, the FollowJointTrajectoryAction, JointTrajectory, Twist and Float64MultiArray for joint position and velocity control. The arms are configured to groups arm_left and arm_right and the head joint is in the head group. The arms have configured limit of 1 rad/s with acceleration of 2 rad/s^2 on the individual joints and 0.1 m/s and 0.1 rad/s for the Twist interface affecting the endjoint. The head has the same limits excluding the Twist as it is

working on single joint instead of multiple. The Twist limits are the same 1 rad/s velocity with acceleration of 2 rad/s^2 for all interfaces.

The sensorring uses the same interfaces as above excluding the Twist and is in the sensorring group. The grippers have interfaces for the FollowJointTrajectoryAction, JointTrajectory and Float64MultiArray joint position and they are in the gripper_-left and gripper_right groups. The grippers are also added to the arm_left and arm_right groups because they work as extensions for the arms and are likely to be used together. The limits for the sensorring and the grippers are the same 1 rad/s and 2 rad/s^2 as for other angular movements.

The base has only Twist interface and the limits are configured to match the existing velocity limits. The linear limit is 0.31 m/s where the original is 0.3 m/s. Acceleration is set to 0.4 m/s^2 to get around a second for acceleration to full speed. The rotation of the base is set to the 0.5 rad/s with acceleration of 0.5 rad/s^2.

6.7 Future improvements

The FollowJointTrajectoryAction and TrajectoryAction on the arm interfaces on the Care-O-bot 4 can trigger hardware faults when the movement is stopped even at slower velocities when using the SLOW_STOP operation mode commanded by the fault topic. As mentioned before, the slowdown functions are not operational on these interface types. The arms need to be recovered from the fault state for them to function again. This could be avoided by not cancelling the current motion, which causes the sudden stop of actuators. Instead a new position ahead of the current movement could be calculated and set as a new goal with zero target velocities. This could lead to the joints decelerating during the SLOW_STOP and not trigger a hardware fault on the arm actuators.

Second improvement also involves these two interfaces, the slowdown feature described before is not working as keeping track of the passed point was not implemented, but the slowdown feature was implemented and tested on with only the joint end position defined on the action interface. The trajectory replacement can be executed using the existing functions checking the limits. Only the tracking of points, removing the passed points and possibly generating an intermediate slowdown point as done in the testing would need to be implemented. The action goal renaming and name mapping needed for the client controllers actionserver is done, but commented out in the codebase as the slowdown functionality was disabled after testing.

The system does not provide any functionality for directly limiting forces produced by the actuators. The parameters are not defined and the limit reconfiguration does not include enumerations for forces. This was not implemented as the Care-O-bot 4 the system is being developed for does not support the ROS effort interfaces. The jointTrajectory messages include an effort component, but the current implementation does not support it. The ability to limit the commanded forces could be added later,

but as a workaround a fault monitor can be used to stop the robot if excessive forces, not the commanded ones, are detected.

7 Resource Allocator

A component for coordinating the resources for the controllers was needed to fulfill the system requirement 2. A central controller, the allocator node, was developed. Using the hardware endpoints, the multiplexer nodes, for the resource allocation was deemed unsafe and difficult to implement due to the communication needed for appropriate operation, as each multiplexer can be in different state and belong into different overlapping groups. Directly requesting the control access from them could lead to issues if communication between nodes is not perfect.

The client-service model for the access request would also need a dedicated node for serving the response to the requesting controller. A truly distributed system could be developed using the multiplexer nodes dynamically assigning one for each request, but this was deemed overly complex. A separate node for receiving and processing the access requests, which would also command and receive data from the multiplexer nodes, was considered a more viable option.

The ROSGuard system is designed for use with multiple commanding controllers executing diverse tasks and the allocator node is the primary interface for accessing the resources on the robot as directly commanding the multiplexers is not recommended. The client controllers or a separate coordinator, such as SMACH [28] overseeing and regulating the executing controllers, use the services provided by the allocator to query the state of the multiplexer nodes and command the requested controller to be connected into the hardware resources.

As with the other components, all the configuration parameters and control interfaces are in the robot namespace, which is passed to the node at launch. Additional parameters are the input controllers and groups the allocator recognizes as valid input for the requests. The updaterate determines the periodic intervals the queued requests are processed. However the input controllers and the grouping are already defined in the multiplexer nodes. Leaving these parameters undefined can be done on the allocator. If either the input controllers or the groups are not defined, the allocator does an autodiscovery on the multiplexer nodes and gathers the controller names, groups and the control topic names from the multiplexer nodes at system launch. The controller names, mux topics and their groups are the only information stored on the allocator node. The state of multiplexer nodes is not tracked and is queried each time the allocator requires information from the nodes. This simplifies the operation as the allocator does not need to keep track of the grouping relations and their overlap on each multiplexer node. As the multiplexer does not store the information, it can be shut down and restarted without affecting the state of the robot. Only requests published during the inactive period are lost.

7.1 Interface

The client sending a request to the allocator uses the access_request.msg from the rosguard_msgs package. These are replied with the access_response.msg after the allocator has processed the request. These messages are passed in the respective topics under the robot namespace at access/request and access/response.

In the request are included the necessary command, INFO, RESERVE, FREE or OVERRIDE, which is passed to the allocator along with the target of the request and the source controller name. The request also needs to be timestamped as requests which have 60 seconds or more offset to the current ROS time are ignored. This was done to protect the access control to only accept request generated right at the time of request. During testing it was found out to cause problems as request accidentally played back with rosbag would go through even if the timestamps were hours into future on simulation time. These messages are not responded and the clients need to implement a 60 second timeout for the access requests.

The response to these requests depends on the type of request and the state of the multiplexer nodes. Possible answers are SUCCESS, ERROR, RESERVED and OVERRIDE. Other data include the source of the original request and the target. Additional information is provided in an information field and a human readable info_text field. The response topic is shared between all the clients so the clients need to filter their own messages. Therefore the source controller name received on the corresponding request is passed back so the related client can identify the responses. Each controller is allowed to have two pending requests queued. Otherwise the allocator will answer with ERROR and TOO_MANY_REQUESTS on the info field. This was done to reduce possible control flood as a misconfigured controller could queue multitude of requests and delay legitimate request beyond the 60 second timeout.

Client controllers must send requests with the predefined, either by autodiscovery from the multiplexer nodes or from the ones defined for the allocator, controller names and targets or the requests are answered with ERROR and either NO_CONTROLLER or NO_TARGET in the detailed information. No queries to the multiplexer nodes are made in these cases.

In cases of successful INFO requests, meaning that the queried resource is free for the source controller to modify, the answer is SUCCESS with EMPTY information field. For RESERVE and FREE the information fields are SUCCESS_RESERVE and SUCCESS_FREE.

For failed commands the INFO and RESERVE will be responded with RESERVED and CONFILCT in the information field, the offending controllers and their reserved targets are listed in the info_text field of the message. If a reservation fails, no resources are allocated to the source of the request fulfilling the requirement 7.

A failed FREE will respond with ERROR and CONFLICT, meaning that at least one of the target multiplexers has been reserved by another controller and the requesting

controller was not allowed to release it. These nodes are listed in the info_text field of the response. Note that the FREE command does relinquish all the resources it is able to and cancels all ongoing or future movements. This is fundamental in cases where the OVERRIDE command has been used as resources need to released by sending a FREE command to the global group 'robot' to free all the resources the controller has. The override functionality on the multiplexers is explained earlier as it is a special operation used to recover the ROSGuard system from fault states.

The OVERRIDE command can only fail if the controller or target do not exist and it is always responded with SUCCESS and OVERRIDDEN in the information field. Even resources reserved by other controllers will be handed over to the overriding controller with their actions cancelled, fulfilling the requirement 8. The controllers which have been overridden will receive a message on the access response topic with OVERRIDE in the answer field, OVERRIDDEN in the information field and the resource the controller has been relieved of in the target field. Using this information the controllers can determine if and what resources have been overridden and then adjust their operation accordingly.

7.2 Operation examples

The following cases demonstrate the usage and operation of the allocator in normal usage.

7.2.1 Reserving a group

A client controller needs access to the hardware to execute movement on the robot chassis. The robot chassis is in the 'base' group on the ROSGuard system and it includes all moving hardware interfaces on the robot base. The controller has input named 'move_test' on the chassis geometry_msgs/Twist multiplexer. The robot is named 'test_robot' and it is used as the ROSGuard namespace for the robot.

New rosguard_msgs/access_request message is generated and timestamped with ros::Time::now(). The command is set to rosguard_msgs::access_request::RESERVE, target is set to 'base' and source to 'move_test' before publishing the message to the target topic on /test_robot/access/request. Figure 7.1 shows the communication generated on the system by the request.

The allocator receives the request and queues it for the next processing cycle. The message is accepted as the timestamp is correct and the source controller and target exist. As the processing starts, the allocator queries the target group for the source controller on the multiplexer nodes and the nodes recognizing the group name reply back to the allocator with their current state. If all the responses received after one second are positive for the control change, a new query is made to the multiplexer nodes with the control change command. Again all the relevant nodes respond back and if there are no errors the control change is accepted and

the client controller ‘move_test’ is responded by sending a rosguard_msgs/access_-response message to topic /test_robot/access/response with rosguard_msgs::access_-response::SUCCESS as answer and rosguard_msgs::access_response::SUCCESS_-RESERVE in the information field. The ‘move_test’ is written into the source field and the target is set to ‘base’. Using these the client can filter the response from the allocator. The control access request reserving the ‘base’ takes minimum of 1.5 seconds due to the communication. This is an acceptable figure as the elapsed time is low enough that the user may not notice it unless closely observing. The resource needs to be stopped when the control is handed over and the 1.5 seconds is likely to be mixed with the perceived operation of the robot. As explained earlier, the upper bound may be 60 seconds if there is control flood the allocator is processing. After the 60 seconds the requests are ignored. The controller needs to abort the operation if it has not received a response within the one minute timeframe and try again.

If the first query to the multiplexer nodes fails due to some other controlling having reserved resources of the ‘base’ group, the second query is not made and the response message is answered with rosguard_msgs::access_response::RESERVED and rosguard_msgs::access_response::CONFLICT in the information field. The conflicting nodes are in the info_text field. The same response is made if the second query changing the input fails on any node and the multiplexer nodes already reserved by the command are reverted back into unreserved state.

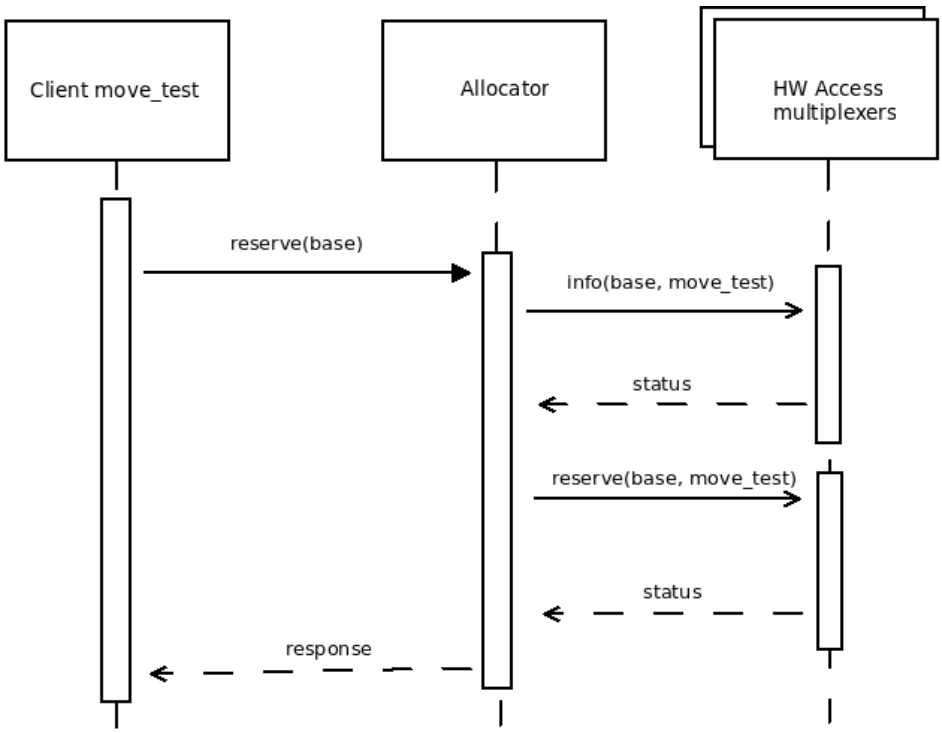

Figure 7.1: *Successful control request.*

7.2.2 Freeing resources

Now the previously reserved resource is released as the controller 'move_test' has finished executing the task and needs to allow access for other controllers. The request message is done as previously, but the command is set to rosguard_msgs::access_request::FREE. In figure 7.2 the interactions are shown.

After the allocator has queued the request, it starts to process it on the next cycle. An information query is sent to the multiplexer nodes and the nodes recognizing the group respond to the query. The info requests received by the multiplexer nodes are reserved to the 'move_test' controller and return without error as the requesting controller has permission to execute changes on the nodes. Next the allocator sends the free requests to the multiplexer nodes and the control change is executed and all movement is stopped, freeing the multiplexer to accept commands from any controller. The allocator then sends message to the response topic with rosguard_msgs::access_response::SUCCESS as the answer and rosguard_msgs::access_response::SUCCESS_FREE in the information field. The response times for freeing the resources is the same 1.5 seconds as for reserving the resources as the behaviour is mostly identical.

The FREE command may fail, but unlike in the RESERVE, the changes are not reverted. All the commanded targets are released if the commanding controller has authorization to do so. Only the controllers that have reserved the

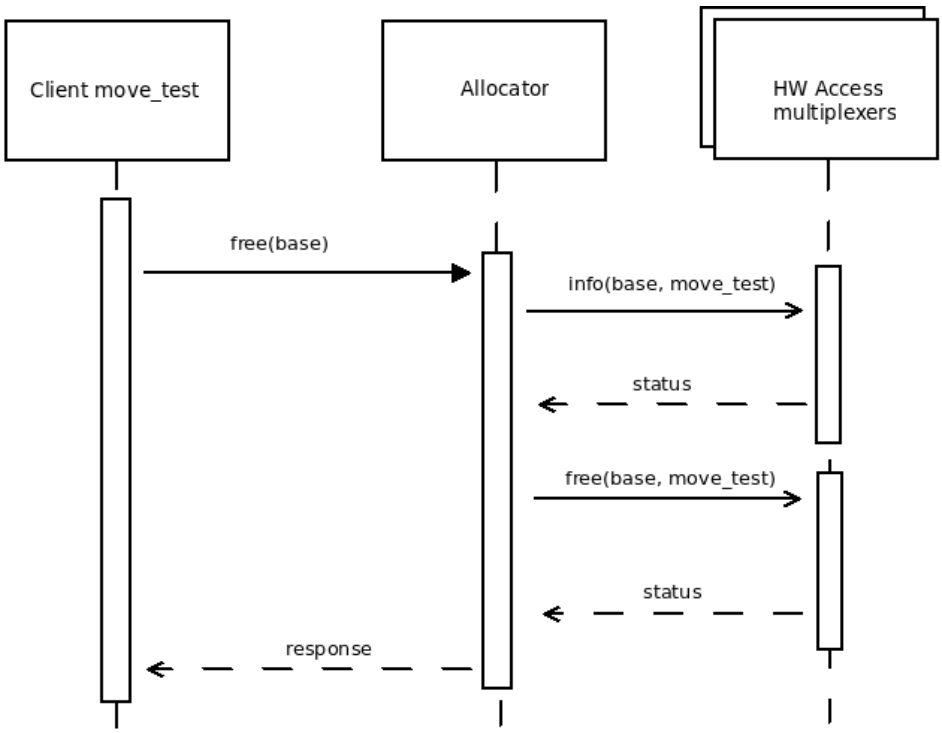

Figure 7.2: *Freeing resources without errors.*

multiplexer node may relieve them. On failure the answer is set to rosguard_msgs::access_response::ERROR, the information is set to rosguard_msgs::access_response::CONFLICT and the failed targets are in the info_text field for inspection. Failure on the FREE command is not critical and is expected in the next example.

7.2.3 Overriding resources

In this example the robot has been executing a navigation task, but has collided with a table as the table edge is protruded and does not show up on the floor level laser scanners. The collision has exceeded the limits in vibration and acceleration sensors and an collision fault has been published on the ROSGuard system as a reaction. This has resulted in a stopped state. The multiplexer nodes and the robot do not move as it is not in normal operation mode. This is intended as the navigation system does not see the table and may collide again with it. In this example the navigation system has reserved a group called 'body' which includes the actuators on the robot base and the torso, which has a single rotating joint tilting the robot. This was done to prevent the torso movement during navigation as it could affect the balance of the robot.

A controller named 'collision_recovery_test' is trying to recover from this fault by navigating backwards, at an angle determined by the closest point on the laser scanners, half a meter and then moving two meters in the direction set by the navigation target, avoiding the previously taken path, before relieving command and clearing the fault. The override mode only allows slow movement as the robot was in stopped state. This is fine for the recovery test as it uses low velocity and the vibration sensors to detect the obstacle it has collided with before. Low velocity collision do not exceed the limits set in vibration and acceleration sensors. This is important as even in the override mode the ROSGuard system will stop the robot if a new stopping fault is detected. It would need to be recovered with new OVERRIDE command to bring the multiplexer back into the slow movement state.

The request to override is constructed same as in the access reservation, but the command is set to rosguard_msgs::access_request::OVERRIDE. The target is still 'base' which only includes the base actuators. The 'collision_recovery_test' has only one output and it is on the base Twist controller multiplexer. The message is published and figure 7.3 shows the resulting communication and actions.

Once the request is being progressed, the override command is immediately sent to the multiplexer nodes as the override command does not execute checks like the other commands. This shortens the normal turnaround for the messages to 0.7 seconds if a second query to the multiplexer is not sent for conflicting resources. Otherwise the response time doubles as new commands are sent to override other resources. The multiplexer nodes on 'base' group change over to the 'collision_recovery_test' but respond to the allocator that the existing controller 'navigation' has been overridden. The nodes also send the previously reserved group name, torso_movement, in the same message.

As resources on the target of the override belong to a different group, it can mean that the previous controller has lost some or all the control it has requested. Therefore the execution of the controller may be incomplete once normal operation is resumed. To prevent this the overriding controller is also given access to all the conflicting resources. A OVERRIDE command is sent to the new group. This leads to the previous controller losing all control over the resources it has reserved and it is forced out so it cannot continue the execution. This is preventing possible issues with the controller resuming execution with partial resources. The overriding controller does not have outputs to the new resources on the torso_movement group, but they are locked at zero movement as there is no input for the collision recovery. The allocator also publishes a response message with answer rosguard_msgs::access_response::OVERRIDE, information of rosguard_msgs::access_response::OVERRIDDEN, the previous group name, torso_movement, and the previous controller name, navigation. Once the previous controller receives this message, it can adapt to the new situation. Overrides can be issued even if the robot is moving, so overriding a resource shared on multiple groups stops the movement on every actuator on the previously reserved group. The controller cannot execute actions with only parts of the originally reserved actuators, thus stopping a possibly hazardous movement.

The allocator receives the final response from the multiplexer nodes of the previous group and ignores them. For a reserved group it is impossible to be partially reserved and broken unless it is released improperly. If broken groups need to be able to be taken over, the code has been left with comments to enable this feature to do more

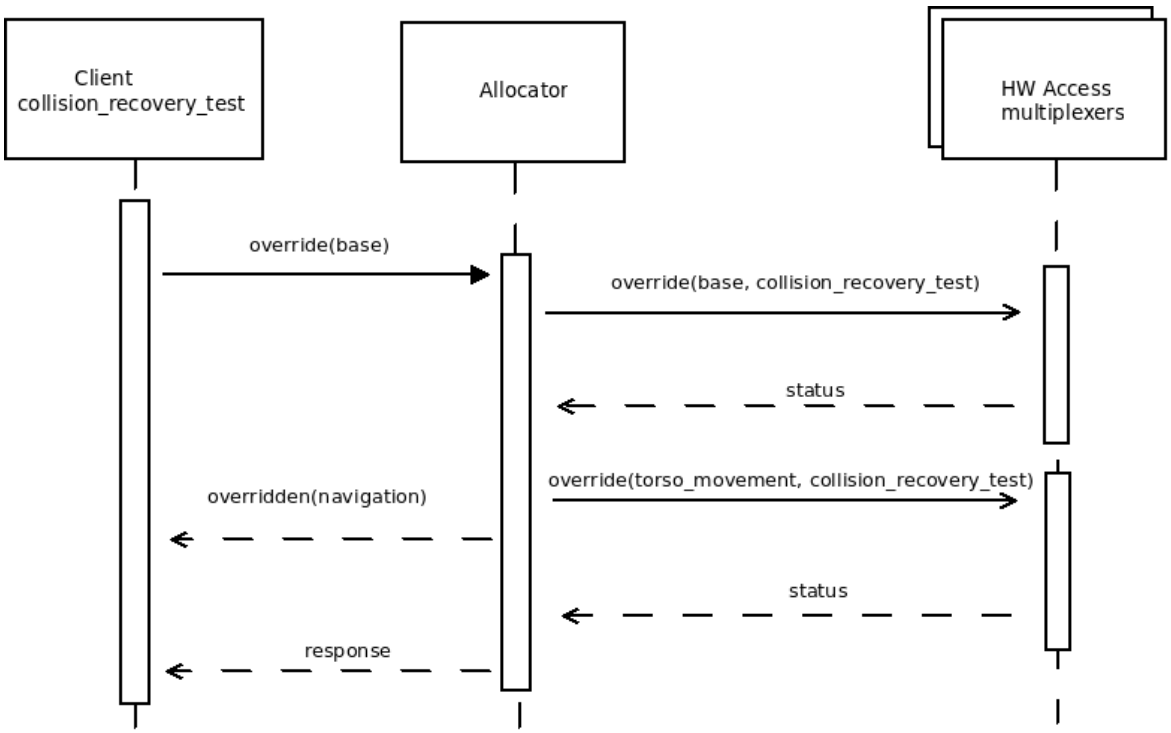

Figure 7.3: *Overriding resources with overlapping groups.*

in depth overrides. The overrides on multiplexers always change mode so there are no further group conflicts to resolve. The overriding controller is notified with a SUCCESS and OVERRIDDEN. In the info_text field the override is declared and the need to use the 'robot' group for relieving the resource is advertised.

The OVERRIDE command always need to be freed with the global group "robot" including all the multiplexer nodes due to the fact it may have reserved other resources than the ones originally specified as targets. This decouples the controller from all the resources, but it should not be an issue as the OVERRIDE command should only be used to recover from faults. The overriding controller needs to be able to take over the targeted resource no matter what state it is in. The additional resources reserved outside the original target are locked to zero movement at the moment of control change.

After releasing the resources, the multiplexer nodes are downgraded back into stopped state as the fault has not been cleared. If the 'collision_recovery_test' has managed to accomplish its goals, it may resolve the fault. If no other faults are present, the multiplexer nodes are set into normal operation mode and the robot may continue normal operation. Possibly the navigation may take the previously reserved resources back and continue towards the navigation goal.

7.3 Access reservation library

Implementing the communication protocol for every controller could be tedious and in most cases, where additional functionality of the detailed error messages are not needed, the access reservation is most likely done by duplicating from the existing examples. For roscpp applications a shared library was developed with some additional functionality to ease the use of the ROSGuard access control. It can be used by adding the rosguard_allocator package to the package.xml in build and run dependencies. The header to be included is rosguard_allocator_request_lib.hpp from the rosguard_allocator package.

Like other ROSGuard components, the name for the robot namespace needs to be passed to the parent node utilizing the library with the 'robot' parameter. Configuration of the controller name is not necessary as the library can extract the name of the parent node. Therefore the controller name for the access control does not need to be hardcoded and it can be redefined in the launch files by changing the name of the node. The ROS node needs to be initialized in the code before the library object is created for the library to function. The library does not run the initialization, but creates the nodehandles if they are not created before the constructor is called [39].

By giving a name in the constructor, it is used instead of the current name of the node for the control requests. This is useful if the controller name needs to be static. The fixed naming also allows a single node to coordinate multiple clients with differing control inputs, for example if a controller has normal operation mode

and a special recovery mode in case of faults or a global planner is coordinating the resource allocation separate from the executing controllers.

7.3.1 Library interface

The library provides four functions, each for the specified access request command, which take the target of the command as the only parameter. The functions are named the same as the commands: info, reserve, free and OVERRIDE. They execute the specified command and return a boolean if the query was successful or not. If the result of the query needs to be inspected, it can be extracted with the getLastMessage() function returning a rosguard_msgs::access_resonse containing the last message from the allocator.

The library also provides information on the overrides the controller has received if it has been replaced. Function overridden() returns a boolean, which is true if the controller has received a notice it has been replaced. Querying the function also resets the value. To extract which target has been overridden, the message containing the information is returned by getOverrideMessage(). It only returns the last message containing the override information if there are multiple overrides for the controller.

Finally the library provides threaded calls of the functions. Normally the functions block the program execution until they receive response or time out after 60 seconds. The threaded functions run the query in the background and other operations can be executed in the foreground. The functions are postfixed with _NONBLOCK and they return nothing. The return value is available after the receivedResponse_-NONBLOCK() function returns true. The value is returned by the response_-NONBLOCK() which behaves same as the blocking calls. These nonblocking calls are useful for example when using ROS QT without threading and the UI needs to be redrawn and responsive. Sending multiple requests with threaded calls is not possible as the library waits for the response or the timeout before allowing new commands to be passed through.

7.4 Future improvements

The lack of python library for easy access allocation leads to the need to duplicate the callbacks for messages on the topics. The easier the system is to use, the more likely it is used and currently there exists no ready codesamples for python. Using the code from the C++ implementation as a base for the python version is easy for the unthreaded example. The communication is done though ROS topics and there are no non standard components used.

Integration of the access allocation into the multiplexer nodes could be seen as a future improvement. The allocator is the only critical component running on a single node. Offloading the allocation to the multiplexer nodes increases the complexity

without guaranteed benefits, but could lead to safer operation if the ROSGuard system is developed to be more dynamic. Currently inputs for controllers cannot be added without restarting the system, which is satisfactory for research purposes, but not applicable if controllers need to be loaded dynamically. A possible workaround is to configure every dynamic controller at launch, but depending on the number of controllers it can lead to cluttered namespaces filled with automatically generated input topics.

The C++ library cannot distinguish multiple calls from each other when receiving notifications of overrides. Only the latest message from the allocator is stored, therefore it is possible to discard the messages before they are processed. The current functionality is sufficient for normal use, but may cause issues if complex reservations are done. Building a queue inside the library should be possible if the complex functionality is needed in the future.

8 Firewall

As explained in section 3.1 the usage of a firewall was originally tested as an access control implementation. During the testing it was observed that blocking individual nodes from communicating with each other was possible. This was developed into a separate firewall node for locking down the resources. The firewall is not intended for security purposes to protect the robot from external threats, but to work with ROS topics and their nodes to prevent accidental communications to the specified topics, thus fulfilling the requirement 3. Usage of the firewall is optional in the ROSGuard implementation. The firewall is not dependent on the rest of the ROSGuard components apart from the messages so it can be used as standalone solution to protect ROS topics.

The firewall node operates on the defined topics by monitoring the subscribers on the local machine with the ROS slave API and then using the iptables to block network communications from remote or local publishers to the local the subscriber. The firewall cannot affect internal node communications defined as intraprocess publishing [40] as it is outside the scope of the firewall capabilities. Starting and stopping of the firewall has been made simple by not implementing a list of approved connections, but the implementation requires the firewall to be started manually every time it is needed. Upon starting the firewall all the connections to the defined topics are recorded and kept connected, only new connections are blocked. This leads to easy configuration of the firewall, but it is at the responsibility of the user to review the connections of the topic. This can be easily achieved with ROS topic tools. The firewall is likely to be used with the rest of the ROSGuard environment, therefore the firewall is best to be configured only for the robot hardware resources where the accidental messages are likely to be published by original code not using the ROSGuard components. The hardware interfaces are also monitored by the multiplexers nodes, which inform if there are multiple publishers on the topics. If these warnings are not present and the firewall is activated for the topics, the hardware interfaces are essentially locked down and only communicate through the multiplexers.

The original operation mode was to lock down the ongoing connections and then block the slave API port on the subscriber node to prevent further communications and new publishers to the node. This proved to be ideal solution for blocking access. Blocking access to the slave API does not interfere with the normal operation of the node on the ROS topics and the node continues to operate receiving messages from the current publishers. This has been tested up to 24 hours of operation. As the locked down mode does not allow any communication to the slave API, the node is unable to receive default ROS service calls as they initiate new connection every time they are called. Using persistent connection [32] for ROS services on the locked node would circumvent this issue as the firewall allows the ongoing connections to stay active. The other issue with locking down the slave API is the fact that the node no longer receives calls from ROS Master, causing the shutdown signals to be ignored. Blocking the ROS Master is deliberate as the negotiation between subscriber and

publisher for a new connection is started by the subscriber once ROS Master informs the subscriber of a new publisher [24].

A protected mode was later added to the firewall implementation to counter these issues. In this mode the slave API is not blocked, the node can receive messages from ROS Master and create new connections to other nodes. As a result the normal ROS service calls works and the node responds to normal ROS events, such as ROS master shutting down the node to prevent naming conflicts if a new node with same name is created. The protected mode does suffer from a slow reaction time, where a new publisher can connect to the subscriber and deliver messages before the firewall can detect and block the communication. The firewall can be configured to publish a fault message in case it detects a new publisher on a protected topic. In locked mode the unapproved connections cannot be tracked as they cannot form in the first place.

If the publisher advertises but does not publish, the firewall in protected mode is able to block the communication channel before any messages are published. When the publisher sends out a message, the firewall rejects the data packet. This happens only in the default operation modes with TCP connections. The ROSTCP is retransmitting the message as the receiving end did not respond to the TCP packets. This retransmission was also observed in the original firewall testing and after 15 minutes the publisher node negotiates a new communication ports for the message delivery. This renegotiation leads to the protected mode to constantly poll the subscriber nodes for new unauthorized connections. ROSTCP is used for internode communications unless the connection is marked as unreliable. In this case the connection is using ROSUDP. As the firewall was built to secure down the hardware resources on robot, the usage of UDP communications, where messages can be dropped by connection errors, was deemed unlikely.

The locked mode is recommended if the subscriber node does not have ROS services or they are unused. The possibility of accidental publishing to the subscriber is minimal and the implementation is far more robust than the protected mode. The locked mode also handles the blocking of unreliable connections as the publishers cannot form the UDP connection in the first place. This also leads to the protection of the ROS parameters on the hardware interface as they cannot be modified due to the slave API being blocked.

8.1 Configuration

The topics the firewall nodes are monitoring are ROS parameters in the robot namespace. The ROS parameters are set in protected_topics and locked_topics. These are space separated listings of topic names in fully resolved format. The topics are robot specific and are best to be loaded from separate configuration file. Configuring the topics in launch files under each node is not recommended as it can lead to desynchronization if robot configuration is updated. Topics can also be added and removed by hand using the firewall controller user interface or by publishing

directly to the firewall control topic. The control topic is shared between nodes and each computer on the robot should have one firewall node.

The node configuration parameters include the updaterate, which is the polling interval of the protected mode, and if the firewall is to send a fault on new connections under the protected mode. At each polling interval the local subscribers are inquired for new connections when the firewall is active. The timing is not guaranteed as each connection to a subscriber node takes time and may fail. This update rate is recommended to be set as low as possible if the protected mode is to be used. By default this is set to one second. The faster new connections can be detected, the better the firewall can protect the hardware resources. Setting this too low results in the polling mode restarting immediately after the previous round has finished, resulting in maximum polling rate. The second configuration item, which determines if the firewall publishes a fault when it detects a new connection on a protected topic, publishes a CHASSIS type fault with OUT_OF_CONTROL as description. This reflects directly to the events where a new publisher is on the robot hardware interface bypassing the multiplexer node. This feature is off by default.

8.2 Technical implementation

The following section details how the firewall operates from the initialization to shutdown.

8.2.1 Initialization

As mentioned before, the firewall node is using iptables for blocking the message at packet level. As iptables has no other interface apart from the iptables command [41], the node needs to run as a superuser and execute the desired iptables command on separate shell. Upon startup the firewall node checks if it running effectively as a root, but also checks if the firewall is running as the root user. This due to the fact that the firewall node is designed to be ran with the executable as effective root with SUID bit set. If these conditions are not met, the firewall will output ROS error message with the necessary commands, including path to the executable, to set the SUID bit correctly. This setup allows the firewall nodes to run with root privileges on any user when included in the launch files.

The firewall only affects the nodes running on local computer and therefore it needs to determine local IP addresses and hostnames. These are acquired by going through the network interfaces and extracting the IP addresses. During the testing on Care-O-bot 4 this method was found out to cause segmentation faults and resulted in the firewall crashing. The CAN driver creates interfaces that can be listed with the network interfaces, but when checking if the interface is a network interface it causes memory errors. This was prevented by excluding all interfaces starting with letter C from the listing. With normal network naming standard, all normal

network interfaces should be listed as they begin with either E, W or L for ethernet, wireless and loopback. Fixed hostname localhost and IP 127.0.0.1 are also added to the list along with the hostname given by gethostname function. The firewall node will be identified by the hostname and it will be included in the console, rosout and debug messages. After this the node creates and flushes the iptables chain named ROS_TOPIC_CONTROL_IN and adds jump to the INPUT rulechain for this. Now two timers are created for the node. The protectionTimer, which calls the polling function going through the subscribers only when the firewall is enabled, and debugTimer.

The debugTimer runs every five seconds and uses the ROS Master API [42] call getSystemState() for acquiring the list of nodes on the system. The list of subscribers is opened and then filtered by the topic name. Topics that do not match either the ones in locked or protected list are ignored. If the topic is in the locked mode, all the nodenames under the topic are added to the locked subscribers list. Likewise the nodenames are added to the protected subscribers list if it is in the protected mode list. These nodenames are sent to the ROS Master with the lookUpNode() call and the acquired host and port are stored for the next step, but only if they reside on the local computer. An unavoidable feature is that the system makes no checks if the same node exists on both on locked and protected lists. The locked mode has higher priority so any node that has protected and locked topics will always work in locked mode.

The previously recorded host and port configurations of the target subscribers are contacted with the getBusInfo() call of the Slave API. This gives all the connections the target node has active. Connection details of the source and destination port, host combination are stored along the topic name, subscriber nodename and the operation mode on the topic.

8.2.2 Firewall startup, operation and shutdown

To ensure the listing us up to date, the debugTimer function is always executed before the firewall is started. This updates the listing on connections, but it is marginally faster and more reliable as it has all the connections older than five seconds already stored. The connections used to acquire the information are not perfectly reliable. Once the firewall is enabled, all the previously recorded host and port combinations are iterated through. All source port to the corresponding destination port pairs are explicitly allowed and the all other connection to the destination port are rejected with icmp-admin-prohibited. During testing it was found out that the nodes ignore REJECT hints so the connections can also be blocked with the DROP rule. Finally the Slave API ports of the nodes with locked topics are locked down with the same REJECT pattern. This is the end of operation for the topics in locked mode.

For nodes with topics in protected mode the protectionTimer calls the scanProtected function of the firewall node. This function scans through the known nodes with protected topics with the getBusInfo() Slave API call and compares the current

connections of the protected topics against the list saved by the debugTimer, which contains the active connections before firewall was enabled. A connection in the current listing not found in previous connections is marked as a new connection. This new connection is rejected with the source port and destination port with the TCP packet having PUSH flag set. Blocking the PUSH messages means blocking the data frames transferred through the ports, but ACK and other frames are still passed between the nodes. For example if there is a publisher that is connected to a subscriber, but not sending any data, the keepalive messages on the connection are still working and from the publishers point of view everything is fine. Otherwise the retransmission of the keepalive occurs and new connection is created as described before. By having the keepalive working, the publisher starts the retransmission only after publishing the first message. After the new iptables rule for the new connection is executed, a debug message is sent in the firewall debug topic containing the connection information. Additionally the firewall node publishes the OUT_OF_CONTROL fault if so configured.

A new subscriber can be added to the system once firewall is running, but it may not have had time to create necessary connections for proper operations. This is not a problem as the configured topics are meant to be used for the hardware interfaces on the robot. If the firewall is enabled, recording and monitoring the topics is recommended to be done on separate computer without a firewall node to prevent the firewall from interfering with the recording.

When the firewall is disabled or shut down, the iptables chain is flushed and all recorded ports are cleared from the memory before the debugTimer starts recollecting them. The node will automatically flush the chains even if it is shut down before disabling the firewall.

8.3 Difficulties

When the system was being developed there were problems with the documentation for the used components. The C++ implementation of the XMLRPC (Extensible Markup Language Remote Procedure Call) used to communicate with ROS Master and Slave API was lacking and required trial and error to extract the needed values. The XmlRpcClient object used for communication was found to have bugs as it has no configurable timeouts, although the functions were defined. In some cases the calls to execute the desired command were left pending indefinitely. This resulted in the need to thread the calling operations on the firewall node and the need to filter out the locked Slave API ports to be never connected as a rejected connection also caused the calls to hang forever.

8.4 Firewall controller

Control of the firewall is done with a separate user interface written in ncurses. It is able to start and stop the firewall, add and remove topics and view state and debug information from the firewall nodes. The separate UI was implemented as the firewall is controlled by topics and once the firewall is up those topics are blocked from outside access. Starting the firewall is possible by using "rostopic pub" command, but after that the control over firewall is lost as rostopic can no longer connect to the firewall nodes. This is the main reason the user should not need to use any of the control topics provided by the firewall directly. Using a separate application for control keeps the subscriber/publisher connections alive. It is also recommended to be launched inside GNU Screen. By spawning the user interface inside screen program, the connection is kept alive while the interface is not used. The user can open screen to control firewall, detach the screen and leave the robot. The robot is likely a standalone unit running its own computer so the control of the firewall needs to be possible even if the operator disconnects from the robot. The interface is shown in figure 8.1 running on Care-O-bot 4 simulation.

Implementing a login method to control the firewall was out of question mostly by the notion of Schneier's Law [43, 44]. Securing it down and verifying it would have been exhaustive. Instead it was opted to rely on screen and the fact that the system has ssh for access. In case the ssh is compromised, the firewall does not matter as the system is not under control anymore. The firewall itself is also a dangerous component as it is running as effective root, so restricting outside access to it only sensible.

8.5 Care-O-bot 4-8 usage

A firewall node is configured in the launch files to be launched on each PC onboard the robot. The configuration for these nodes is loaded into the robot namespace with rosparam using a YAML file containing the names for locked and protected topics.

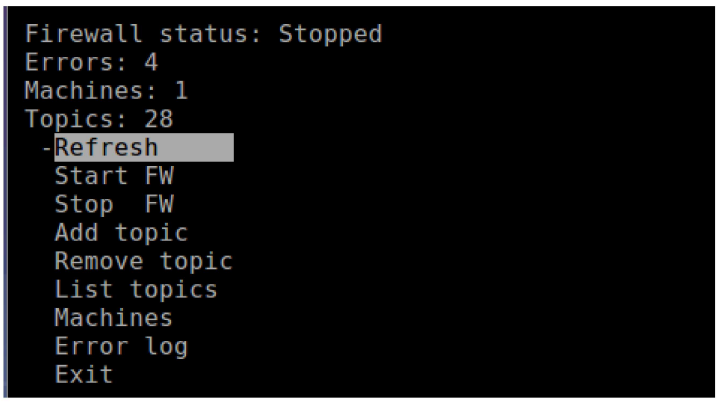

Figure 8.1: *Firewall control interface.*

The launch files also include the firewall controller interface, which is launched inside a detached GNU screen session on the base computer. To access the firewall interface an ssh session to the base computer is needed. The command "screen -r firewall" brings up firewall controller. The commands to set the firewall node run as a root user need to be ran on the base computer only as the network filesystem is used to attach the ROS workspaces on other computers. The current configuration allows the SUID bit on mounted filesystems, but if this changes, the errors on firewall controller will indicate if the firewall nodes are unable to execute the iptables commands on the other computers.

The configured topics are all the hardware resources associated with the multiplexer nodes. All the topics are in the protected mode of the firewall as the current implementation needs the ROS services of the target nodes available. Initialization and recovery of the hardware is done through these services and using the locked mode would cause the current cob dashboard to be blocked from sending recovery and initialization commands. Using the protected mode leads to the possibility of commands going through to the protected topics for several messages before the firewall detects the new communication and blocks them. If the dashboard is modified to use persistent service connections or another node is developed to take care of initialization and recovery, the topics can be changed to use the locked mode denying access for any new publishers.

One known problem is the screen session not activating the ROS node of the interface properly. Attaching the screen session wakes up the ROS node and it begins to work normally. If the screen session is not attached, the node does not automatically die if roslaunch is stopped or a ROS node with same name is created. These bugs only affect the old control interfaces if the roslaunch including the controller is ran multiple times. The latest controller spawned by roslaunch always works.

8.6 Future improvements

Currently the protected mode polls the subscriber nodes for new publishers and their connection details periodically. The polling method is also sequential, which could lead to delays and exceed the polling interval defined in the launch files. The polling interval is already exceeded if a node fails to respond on the ROS slave API, which happens occasionally as explained before. The delays and use of resources are not ideal. The protected mode needs minor rework for proper implementation, but as the protected mode was to be used for nodes with ROS services without persistent connections, the implementation is adequate and works as intended.

To fix these issues the polling mode could be converted to only react to new publishers. This would reduce the number of connections created to query the nodes for new publishers, therefore reducing the frequency of failed connections. The sequential nature of querying the nodes would also be removed, leading to faster operation. Only the subscribing nodes associated with the topic for new publisher would be

inquired for connection details with ROS slave API. The implementation should be possible by using the ROS master API. ROS Master has information on nodes and their subscribers, publishers, services and their topics. New publishers to a topic inform the ROS Master of their connection details. Then the subscribers receive this information from the ROS Master and connect to the slave API and negotiate direct connections for delivering the topic message payload. This could be exploited by creating a fake ROS node using the XMLRPC for subscribing into each protected topic. The ROS Master will inform of new publishers to the fake node and the information on topic, node and its ports are saved. Using this information the subscriber nodes with the same topics could be queried for the connection details of the new publisher. Then these new connections would be firewalled.

The connections to these topics still have to be checked periodically as in the original polling mode 15 minutes after firewalling because a new port may have been negotiated between the publisher and subscriber. After a new connection has been established, it can be firewalled and the old port can be released. This is to be repeated every time until the publisher is no longer advertising the topic. This solution is not ideal as it still needs to poll the nodes, but it is faster than going through every subscriber.

Final improvement for the protected mode would be to include UDP connections under the firewall. UDPROS communication is designated as "unreliable" in ROS as it does not use TCP to ensure successful message delivery. As the firewall was built to secure the hardware interfaces on robot, the UPD communication was not a priority. As the UDPROS messages do not check for successful delivery, the protected mode should not time out after 15 minutes. Using the UDP in protected mode should therefore lead to less bleedthrough by the firewall as it does not create new connections. This does not affect the locked mode as in locked mode the new publishers cannot connect to the subscriber slave API port, this connection is always performed with TCP transmissions.

All of the previous improvements have involved the protected mode of the firewall. The mode was originally added for compatibility with normal ROS services without persistent connections. Getting rid of the mode and making the locked mode work in all situations would be ideal. This could be possible by using the firewall to redirect the Slave API communications of the node from the ROS master to a filter. The master informs the subscribers of new publishers and their topics. By filtering the firewalled topics from the messages before relaying it back to the subscriber node, the topics should be secure. This implementation allows the non firewalled topics, services and other ROS functions, such as parameters, of the node to operate normally.

9 Monitor interface

The state of the ROSGuard system is based on the state of the multiplexer nodes. The operation modes are controlled by the faults in the system and the developed monitoring interface subscribes to the faults along the debug messages from the multiplexer nodes. These messages are recorded and can be inspected from the UI.

The UI is written as a series of dynamically generated HTML pages. A HTTP server constructed with async_web_server_cpp [45] ROS package serves the pages to the browser connecting to the monitor node. The node is required to run on a computer onboard the robot for it to receive the data reliably on the mentioned fault and debug topics. The node can also publish into the fault topic as the UI is designed to be used as a software emergency stop. The pending faults are integrated into the monitor nodes and they are published at the specified times unless they are revoked before the timeout.

9.1 Faults

Each fault received by the monitoring interface is listed and can be inspected. Resolving a fault is possible from the detailed information page of the fault in question and the node sends the fault back into the topic with the RESOLVED enumeration, removing the fault from the multiplexer nodes. In the listing the faults are color coded according to the effect on the multiplexer nodes. As seen in figure 9.1 faults not affecting the state of multiplexers are colored green, magenta for slowing faults and red for faults commanding the robot to stop movement. This provides an easy route for inspecting the faults affecting the movement of the robot. The pending faults are shown in the same page along the seconds to the activation of the fault. The pending faults can be inspected and cleared like the normal faults by clicking the link to the detailed information page.

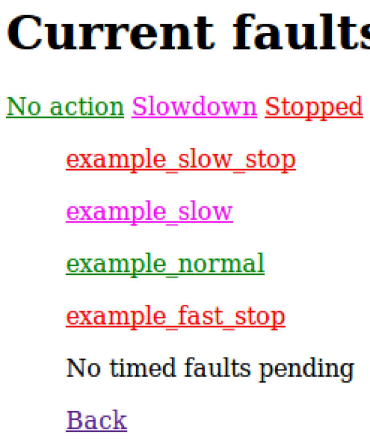

Figure 9.1: *Fault listing on HTTP interface.*

The UI can be used to generate faults to test the system by injecting faults manually. As shown in figure 9.2 all the fields in the faults are configurable except the timestamp. The identification can be set manually, but if it is not configured, the UI generates one by appending the current ROS time to the end of the component name. The software E-stop functionality uses the same backend as the manual fault generation, but the UI is three buttons for the each desired state of the ROSGuard system. The backed accepts input with HTTP GET forms and generating custom buttons or links for testing specific responses from controllers to faults is straightforward.

9.2 Multiplexer

As the system includes debug information intended for the end user, the monitor interface includes a component to display this information. The multiplexer debug information presented before is filtered for display. The listed debug information are the undesired events in the multiplexer nodes and the normal operation of the nodes are not shown. The UI is not meant to query the current state of the nodes for example to extract current active controller. This is to be done with the allocator INFO commands.

One of the presented states are the commands in inactive mux inputs. A controller outputting commands while the multiplexer is not switched into the input tells the controller has not properly reserved the resource, or it has been overridden. Unauthorized access to the control interface. For example a controller releasing resources it has not reserved. This is normal to see for overriding controllers as the rule is to use the global group to free resources. Velocity or acceleration limits exceeded messages can point to a misconfigured controller commanding rapid movements. The limits can be exceeded by any controller when the faults have triggered a state in the ROSGuard system slowing the robot down. These cases can be inspected easily as the debug messages includes the state of the multiplexer node and if it is not in normal state, these messages are safe to ignore. The final message is for guarding

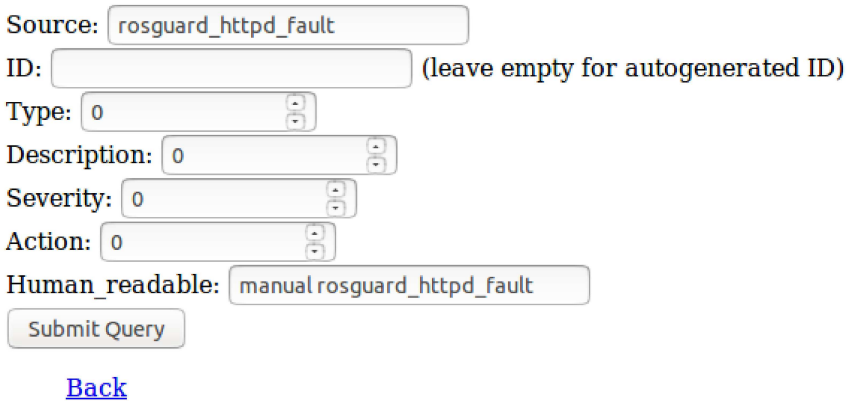

Figure 9.2: *Fault generation on the HTTP interface.*

the output for the hardware interface. Multiple inputs on hardware topic could mean that the multiplexer node has been bypassed, possibly leading to dangerous situations if no movement is allowed or there are conflicting commands published to the node commanding the hardware. In figure 9.3 is an example on the Care-O-bot 4 simulation. There are multiple warnings of multiple inputs on hardware node. This is due to the teleoperation node directly publishing to most of the hardware interfaces. For the base movement the existing priority multiplexer and velocity smoother are connected between the hardware and the teleoperation.

9.3 Future improvements

The enumeration of different faults is not shown in the user interface of the fault information. Only the numerical values are shown and the values needs to be looked up if the human readable field is not set. This is more pronounced in the fault generation interface, where the user needs to input the values manually. This could be improved by looking up the enumerations and displaying them. The improvement is not critical, but improves the usability. Extracting the enumeration names would need to come from the original message files as acquiring the information from the C header is impossible with the C++11 standard used and due to the fact that during the build the automatically generated C message header has reordered the enumerations in the message. The monitor node could be expanded to load the message file for the faults and parse it, generating the necessary data structures to output the mapping of the enumerations.

The visualization of different multiplexer nodes is also difficult. Each node is displayed by their identifier, the topic or action name, but in reality some of the interfaces are grouped into one physical actuator. This grouping is not evident in the interface and may cause confusion. As the system was designed for generic approach, the UI cannot be customized for each robot. Functionally the monitor is fine, but could

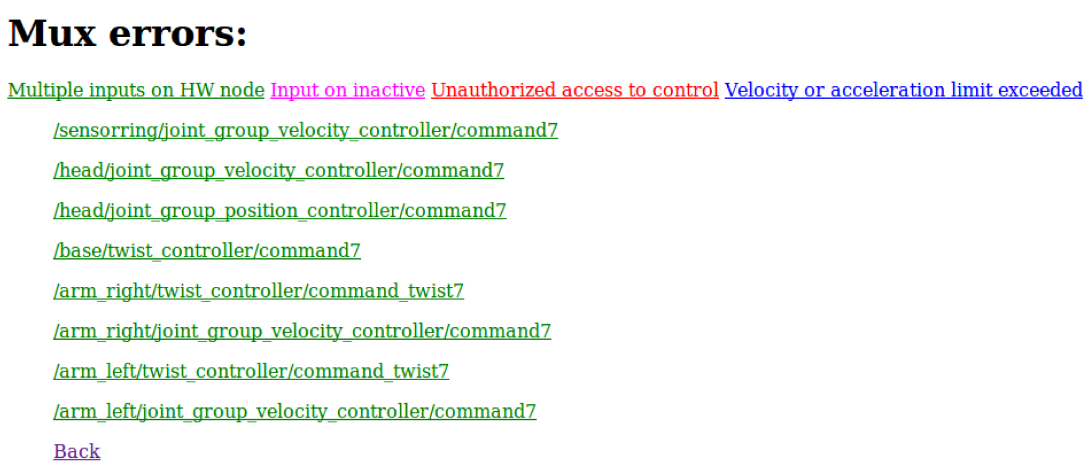

Figure 9.3: *Multiplexer debugging information on the HTTP interface.*

be refined with including a robot specific configuration for the grouping, but falling back into original listing if none is loaded.

The possibility of adding a “clear all faults” button was considered. Adding it for every robot could lead to issues as the operating environments of robots are vastly different and the feature may see abuse if all faults are cleared every once in a while instead of actually resolving the faults. A configuration option could be added to turn the feature on and off.

10 Testing and use case examples on Care-O-bot 4

This section contains the description of some of the components used to test and operate the system during the development. Later some of these components are used to show example use cases on how the system can be utilized on the Care-O-bot 4 number 8 seen on figure 10.1.

Figure 10.1: *Care-O-bot 4-8.*

10.1 Example components

During the development the individual components were tested with scripted input messages for interfaces and other components. The actions and outputs were verified, but the interaction between the components needed to be tested. The simulation environment for Care-O-bot was heavily used to test these combinations. Simple test programs were developed to execute different actions on the platform to observe and determine if the behaviour of the system was correct. These components can be used as examples on how to construct different components for the system.

10.1.1 Cob4 base dummies

The base dummies are a collection of three controllers where each controller publishes a twist message with different value on the angular Z axis, rotating the robot around the base. The controllers utilize the access allocation for reserving the resources, testing requirements 2 and 7. The outputs are routed to the inputs of the multiplexer on the robot base, conforming to requirement 1. One of the controllers also includes a randomized override so it can take over the resource even if it is not free at the time, testing the requirement 8. All of the controllers can detect if they have been overridden and stop the execution of the current action upon receiving the override notification. However they do not utilize fault interface and will continue normal operation regardless of fault messages. The difference between each controller is the access reservation method.

The three possible examples are included in the codebase. First is the constructed with manual subscriber and publisher to the access control topics and callback function to monitor the response. This could be ported to Python without issues. Second one is the usage of the access reservation library in the normal operation mode for controlling the access requests. The final one is the example using the threaded calls in the library for managing the access.

10.1.2 Arm collision monitor and resolver

The simulation environment spawns the robot in the default state and the arms of the robot are extended to the sides with zero angle on the joints. If the arms are not folded to the side of the robot, they will collide with the environment when the robot is navigating near the walls. The built in collision avoidance by the laser scanners on the velocity smoother only keeps the base of the robot from colliding with the obstacles detected.

The arm collision monitor was built as an example fault monitor and it was used to test the effects on faults on the simulated system where the robot is operating autonomously. The constructed fault monitor reads the joint states of the arms. If the joint angles are near zero, the arms are assumed to be in home position and the

monitor is active for the relevant arm. As the majority of collisions affecting the arms came from the walls on the simulated environment and they extend from the arm level to the floor, the node is using the laser scanners on the floor level to detect the areas under the arms. If it detects enough points in the monitored area, a configurable fault is published, testing the requirements 5 and 6. The fault can stop the multiplexers either with fast or slow stop, additionally the node can be configured to only publish the fault without affecting the multiplexer states. The fault type is ENVIRONMENT with COLLISION description. The severity is marked as ROBOT_HARM as the collision could damage the arms or the grippers. The information on right and left arm collision is encoded into the data field. The identifications of the published faults are saved and the node will not publish new faults before the old ones are resolved.

The arm collision resolver is direct counterpart to the collision monitor and only processes faults published by the monitor node. The resolver overrides the relevant arm if no other controller has reserved it and folds the arm to the side of the robot. The override enables the target actuators as by default the monitor node commands the multiplexers to stop the robot and therefore test the requrement 8. The fault is resolved once the arm has finished the commanded trajectory. If the arm is reserved or cannot move and times out after 30 seconds, the fault is not resolved and the resolver makes no further attempts to resolve the issue before new fault messages are received. The node does not save the faults and the fault will be left active if the resolver fails.

10.1.3 Navigating in the test environment

In a basic operation test the robot is spawned in default state into the ipa-apartment [46] simulated environment. The ROSGuard environment is brought up from the cob4_base_dummies package launch file base_nav_test.launch, which configures the system. Controllers for the base and arms are defined for the base dummies and the arm collision resolver. The base also includes a Twist proxy where the ROS DWA (Dynamic Window Approach) planner included in the Care-O-bot codebase is remapped to. The multiplexer on the base is directly connected to the hardware command topic. The priority mux and the velocity smoother included by Fraunhofer are bypassed so they do not interfere with the testing.

As the testing environment is brought up, the base dummies start reserving the twist controller commanding the wheel actuators. They will spin the robot around for seven to ten seconds before releasing the resource, at which point the other controllers may take over. The cycle is continued indefinitely. Once a command is given through the navigation goal for the DWA planner, it starts to output Twist messages for the proxy. The proxy tries to reserve the resource and usually succeeds on first or second time after the base has been released by a dummy controller. The proxy has a delay when it tries to reserve after a failure and another controller may end up taking the resource before the proxy can re-request the access. Once the proxy has access, the DWA planner will drive the robot to the given goal position. Usually

the goal position is put into places where the robot needs to go through doorways and corridors. This brings the arms of the robot into close proximity to the walls and once close enough, the arm collision monitor triggers a fault warning of the possible collision on the arm. The fault triggers the multiplexer nodes to stop the robot movement.

Once the fault is received by the arm collision resolver, it issues an override on the relevant arm, folds it to the side and resolves the fault. As the fault is resolved, the multiplexer nodes go through the stored faults and detect none affecting the operation mode. The normal operation is resumed. The robot continues the path commanded by navigation and in case of doorways, the second arm goes through the same process. A example of the navigation is shown in figure 10.2. The robot can handle multiple controllers accessing the resources, stop before a collision, recover the robot to state it can continue and finish the navigated path as demanded by requirements 1, 2, 5, 7 and 8.

This was executed with multiple permutations with different components issuing overrides, manually published faults from the HTTP interface for the requirement 6 and with the firewall in locked and protected modes testing the requirement 3. Multiple bugs were caught as not all the cases on individual component level were tested properly. For example with controllers having override access to a multiplexer, receiving a fault that is being resolved by some other controller issuing overrides led to the multiplexer state being left in the lowest level even when the fault was resolved. This was due to the special case of override mode being able to elevate the multiplexer from stopped state. To take over multiplexers in normal operation state, which is possible if the override is not used to recover the robot from fault that has stopped the robot, was not considered during the component testing.

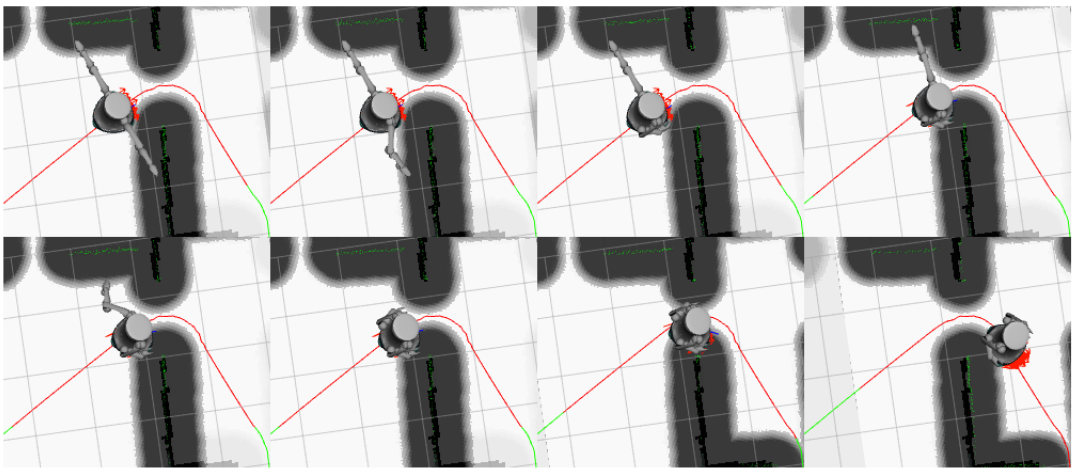

Figure 10.2: *Care-O-bot 4 navigating through a doorway.*

10.2 Use case examples

The examples on the robot use some of the components presented previously and additionally a simple controller coordinating all the movements was developed. The developed controller is configured from the launch file to reserve resources and execute movement in linear fashion. The controller can react to faults by executing a specified movement. Each time a movement is executed an access reservation is made. This slows the controller down, but enables it to do override to recover from a fault. It also recognizes if it has been overridden and aborts the execution. The controller is named rosguard_playbacktest and the commands for the movements are pointed to these inputs on the multiplexer nodes.

The controller takes five parameters which are linked together. First is the movement command, either a ROS bag file or a shell script. The target group or topic of the access reservation. The behavior of the access reservation, when to override and and release the resources. Final two are the execution time of the movement before the next one is enacted and the response to a fault for each movement. If no response to fault is specified, the recovery is not done and the controller continues normal execution. Originally the controller was supposed to only play recorded bag files, but the actionservers used for arms made this impossible. The actionservers need unique identifiers for the goalID parameter and during the bag playback the sequence numbers of the actions are not the same as recorded. Using the rosbag to move the arms was not practical so the controller was adopted to run shell scripts. For the arms these scripts generate random goal identifiers and use rostopic to publish the messages.

A set of launch files were added to bring up the ROSGuard system. Each actuator group has its own launch file, which includes multiplexers for all the provided hardware interfaces. These were gathered into two launch files consisting of the whole system. These include the actuators and the supporting components. One for the simulated environment and the other for real robot, where the nodes are distributed across the computers available. The multiplexer grouping is defined in the two main launch files, but the controller names are defined for the launch file separately. The names are used to automatically generate the inputs for each multiplexer. The desired configuration is defined in the experiment launch file, which includes the ROSGuard system. Switching between the simulation and real robot is straightforward and is done by changing between the two the included launch files sim.launch and robot.launch. An example configuration for launch file is shown in figure 10.3.

As the system was designed to be easy to use, the steps for adding controllers to the system needed to be minimal. The insertion of a controller can be achieved with three steps if the controller outputs the message types supported by the ROSGuard and robot. First the controller is assigned a name and it is added to the relevant hardware groups as an input controller. Second the outputs of the controller must be mapped to match the inputs generated for the controller. Last the controller must

support the access control system either through the controller itself, a central task coordinator or by a proxy providing the access control services.

10.2.1 Pouring tea

As the service robots are not yet widely in use in domestic environments, the possible use case scenarios needed to be thought out and be realistic in nature. A study [47] completed 2011 in UK shows that preparing tea was the overall most requested task for a personal care robot. This task was narrowed down to the most dangerous phase, when the robot is in possible interaction near the human who the tea is served for.

The tea is poured from a teapot into a cup. The pot is tilted forwards until the cup is full and then retracted. If the robot is disturbed during the pour, such as the robot is bumped and the tea misses the cup, the pour is to be aborted and the pot is retracted back into the standby position. The purpose is only to demonstrate the possible functionality and all trajectories are preprogrammed.

The trajectories for the arm were formed with MoveIt! Movement Planning Framework [48]. The right arm holds the teapot and the pour must be started from a position the pot is level. The robot was assumed to be starting from the home position, arms extended to the sides of the robot. The right arm was brought into the level pot position. Once the tea pour was started the pot was brought forwards and tilted for the water to flow out. Once the cup was filled the pot is leveled and brought back towards the robot. The pouring state was ten seconds long and the tested faults and other actions were mostly inserted during this time. During every movement the playback controller was set to recover to the level state with faster movements than with the normal stopping of the pour.

The inserted faults were introduced via the HTTP fault interface and the arm collision monitor, which was configured to not stop the robot. One test was to introduce control requests with the false “navigation” interface on the base of the robot. All of the tests and development were done on the simulation environment before moving on the real robot. The robot can be seen imitating the pouring of tea in figure 10.4.

```
<arg name="robot_ns" value="cob4_8"/>


<!-- Load firewall rules for rosguard in correct namespace -->
<rosparam command="load" file="$(find rosguard_demonstrations)/config/firewall.yaml" ns="$(arg robot_ns)" />
<!-- Load rosguard (simulation) -->
<include file="$(find rosguard_launch)/robot/cob4-8/sim.launch">
  <arg name="robot_ns" value="$(arg robot_ns)"/>
  <arg name="base_controllers" value="navigation twist_proxy_navigation rosguard_playbacktest"/>
  <arg name="arm_left_controllers" value="door_arm_resolver"/>
  <arg name="arm_right_controllers" value="door_arm_resolver rosguard_playbacktest"/>
  <arg name="head_controllers" value=""/>
  <arg name="sensorring_controllers" value=""/>
  <arg name="fw_fault_on_protected" value="true"/>
</include>
```

Figure 10.3: *Example configuration for launching ROSGuard in an experiment.*

During the uninterrupted runs the controller successfully executes the serving of tea, testing the requirements 1 and 2. Publishing acceleration faults with the HTTP interface causes the controller to abort the pour and the pot is brought level for testing the requirement 8. During the pouring stage while the pot is tilted the controller has been configured to reserve the whole robot so moving other actuators does not disturb the ongoing task. Trying to reserve the base with override using the "navigation" controller resulted in the control to be passed over to the overriding controller. The teapot was left in the pouring orientation as the playback controller no longer had access to the arm and had aborted the execution. This is the intended result as the overriding controller took over the group in the base of the robot and the playback was aborted as it had lost access to the needed resources. As stated before, the overriding controllers must be able to take over the robot in unknown, possibly hazardous state. The requirements 7 and 8 were fulfilled as the "navigation" had authorization to use overrides for the fault recovery, which in this case was faked.

Once faults configured with messages to stop the robot were published, the observations between the real robot and the simulation started to diverge. If the arms were in motion while the faults were published, the multiplexers cancelled the current goals executing on the arm. In simulation this only resulted in the arm stopping all movement. On the real robot the sudden stop of the actuators caused hardware errors and the actuators were disabled. Issuing the recover commands from the cob_command_gui [49] re-enabled the actuators and they could be operated again. The faults from the arm actuators also appeared in the HTTP fault monitor listing as the diagnostics to fault node automatically translated them. The same also occurs

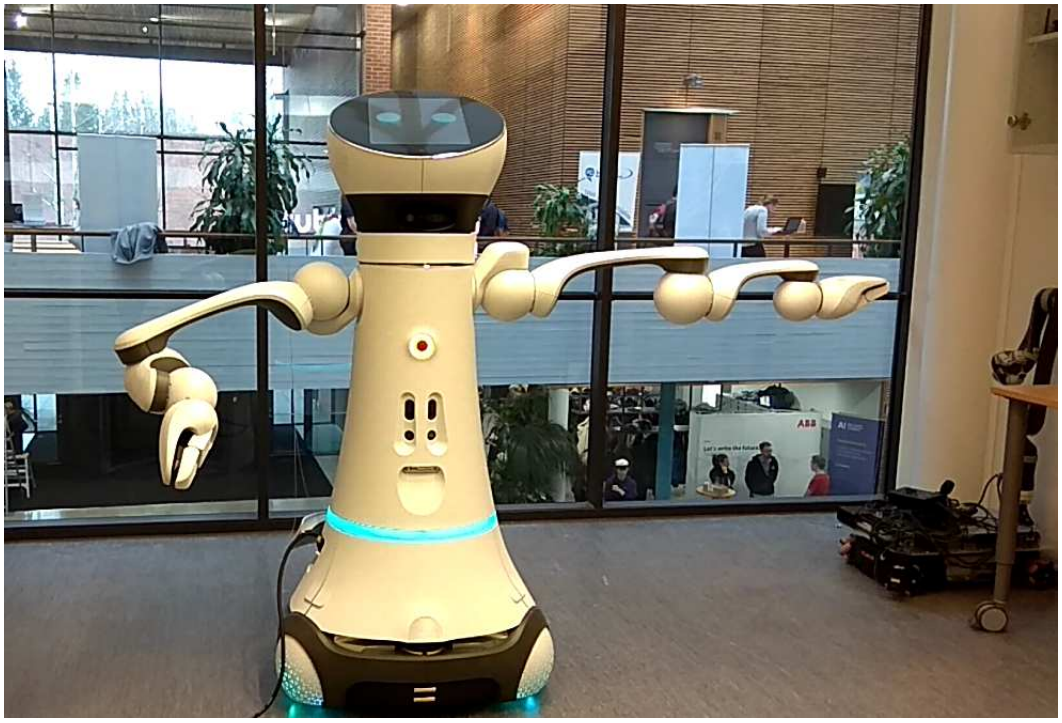

Figure 10.4: *The robot pouring tea from simulated teapot.*

if the override is issued during the motion as the multiplexers do not allow moving actuators to be passed for other controllers. In figure 10.5 the robot has cancelled the pour as a human has entered the operational area. The door arm monitor was used to detect the object, the foot, under the arm.

Other issues during the real robot testing were caused by assumptions in the development phase. The firewall and CAN interface incompatibility was discovered as soon as the tests were started on the real robot. The firewall nodes running on computers with CAN interfaces would crash immediately at system launch. The firewalls would also disregard some nodes as the hostname did not initially match the local machine. The base ROS system running the robot is started at boot and the network connections have not yet formed. Once all the computers have booted and connected, the new nodes spawned on the computers may receive different hostname used in the XMLRPC communication. This was fixed and now the firewall nodes extract all addresses used by the local system. The final issue caused by the assumptions were the automatically generated node names. ROS does not allow nodes to have certain special characters in the names and some of the ROSGuard components used the computer hostname to automatically name themselves if no names were specified for the node. The computer hostnames included forbidden characters and the code was modified to filter the special characters so the name generation would work.

Initially the demonstrations would not run on the real robot at all as the resource allocator rejected the access requests sent from the demonstration controllers. The

Figure 10.5: *Tea pouring aborted due to interference.*

requests were expired. The fault was not found on the developed system, but on the robot itself. The clocks on the robot were not synchronized and were 140 seconds apart at worst. The ROS environment on the Care-O-bot is spread across six different computers. The grippers include two more, but they are not connected to the ROS environment and only accept commands for operating the grippers. As the testing was started, the robot had not been used for other applications as it had been recently delivered. The robot was connected to the Intelligent robotics VLAN (Virtual Local Area Network), which includes the laboratories and the personnel offices. Restrictions have been applied on it so no outside connections can be used to affect the computers in the network. This setup also blocks the NTP (Network Time Protocol) from operating on any other time server than the ones provided by the Aalto University network services. The robot was misconfigured as it was set to only use German NTP servers. A local server had been configured so the clocks would synchronize internally on the robot, but it was disabled and did not work when enabled. As the robot could no longer connect to the German servers and the internal synchronization was defective, the clocks started to drift. This caused the issue with expired access requests. The issue was fixed by configuring a server on the base computer of the robot to be used as a primary NTP server for the other computers and the Aalto server as secondary. As the robot is started the base computer is not immediately used for the clock signal as NTP server does not allow the time to be served before the clock has stabilized. This issue also indicates that the ROSGuard system cannot handle significant changes in the system clock as the ROS time is used for calculating the velocity and acceleration limits. With working NTP setup this is not an issue.

The pour demonstration yielded the desired outcome by exposing the difference between the simulation environment and the real robot. The playback controller is able to pour the tea and react to the external stimulation inserted into the system.

This example only covers some of the hazards that may occur during the operation of the robot, mainly the spillage of hot water and the operator getting too close to the robot. Other hazards to be monitored could be the overflow of the cup and the attachment of the pot to the robot failing. Even pressing the E-Stop on the robot could lead to the hot water to be left flowing causing further injuries and by that assumption it would be safer to let the controller recover to the non-pouring state before an emergency stop is issued. This causes issues with the safety standards as they require the E-Stop to be activated in certain situations, where leaving the robot actuators at the current orientation may lead to further harm. This was also noted by other studies on robot safety [35]. This could be avoided by the requirement of the inherently safe design required by the ISO 13482:2014, but as a personal care robot the task required by the end user may be too generic in nature to be safely designed. The personal care robot may be asked to fetch or pick up items for the end user and these items could be inherently dangerous for the robot or the user. Requiring the robot to only interact with objects it recognizes could limit the usability to unsatisfactory levels. The broader the functionality the consumer robot is, the more likely it is to be integrated into everyday use as the users do not need to adopt their behaviour or environment too dramatically. Historical studies have

proved this [50]. The processing power, perception and artifical intelligence have advanced and it could be possible for the robot to determine the safest action to execute with further reasearch.

The hazards after the liquid has spilled should also be addressed. An environmental warning may be issued through the faults and logged. Addressing the issue may be left to a controller responsible of cleaning the residence. If a human is harmed due to the spillage, the fault should be flagged with the HUMAN_HARM severity. If the robot is a remote caretaker [51], the messages may be passed to the supervising party, who could take over the robot in teleoperation to observe the issue and alert the relevant parties if the client needs help.

10.2.2 The robot opens a water tap

A possible example on how to use pending faults is presented in this use case. The example was not executed on a real robot as the fault messages are faked and the robot does not perform actual movements.

The robot was instructed to open an irrigation system for watering the lawn in the backyard and leave it running for an hour. During this time the robot should to continue cleaning the house, which was the second most requested feature in the 2011 study [47]. During the cleaning the robot collided with a floor lamp, which fell over and confined the robot into the room it was cleaning. The robot was unable to close the irrigation system, resulting in over watering the plants and increased water bills.

The controller responsible of directing the robot should recognize that the command it was asked to do was for a specified time. The planner queued both tasks, the opening and closing according to the instructions received. Once the task for opening the valve come up, the robot published a pending fault for the possible environmental hazard. The fault was timed for the one hour with ten minutes added for the possible delays from task switching and travel times. Then the robot opened the irrigation system and returned to the previous task, which was cleaning the house. Note that the fault must be published before the action leading to possible hazard is taken to ensure the sending does not fail if the controller crashes as it is opening the tap. Even if the controller crashes after publishing the pending fault, but before opening the tap, the controller fails in a safe manner.

The robot continued cleaning, but as it was leaving one of the bedrooms it could not get out as a lamp had fallen over blocking the doorway. The navigation controller tried to find an alternate route out of the room, but failed. As the navigation could not reach the desired goal, it notified the other components it was stuck. This was then tried to be resolved by contacting the owners in the house by the speech interface. The owners were not inside the house so the robot was still stuck. The robot being confined into a room is not considered a hazardous condition so the supervising party, possibly external service sold along the robot or a cloud based interface for the owner, was not contacted.

As the navigation cannot find a route out of the room, the queued task for closing the irrigation system was left in the task queue. However once the timed environmental hazard fault was published, the robot tried again using the speech interface to warn the owners. As the robot received no replies, it activated the connection to the supervising party as the hazard was deemed considerable enough for the service to be used. The owner was notified of the event and decides the proper course of action. This is a test for the requirement 8 where the faults are not the direct result of the current action.

This again brings out the question if an autonomous robot should execute tasks, which may lead to environmental or other hazards. Again this could be avoided by the inherently safe design by using a solenoid valve controlled by the home automation system. However operating a lever, the water tap, is a task the robot should be capable of as the difference between door latches, which the robot should be able to operate, and the tap is not extreme. Again the same issue presented in the previous example emerges. The end users do not want to modify their environment more than necessary and this could lead to tasks which may be hazardous.

11 Conclusions

In this thesis a system for interfacing multiple controllers for robot hardware with additional fault tolerance was introduced and implemented. The system is intended to prevent the unintentional motions from happening on personal care robots when the robot is working with multiple controllers dedicated for executing different tasks. An error in a controller can lead to situations where it commands motions when the connected actuators need to be stationary or a controller can command zero movement in a waiting state while another controller is trying to execute tasks. These motions could endanger the robot, environment or humans interacting with the robot, which goes against the current safety standards for human and robot interaction.

The system cannot replace the conventional methods for ensuring the safety for humans, environment or itself. The active controllers on the robot can still command the actuators according to their programming, which may or may not be faulty. This problem can be mitigated by the secondary requirements set for the system. The ability to slow down or stop the robot independently from the ongoing actions provide the system designer with tools for protecting the robot, environment and interacting personnel from harm.

The configurable velocity and acceleration limits also provide active protection in case the actuators are commanded to move too fast compared to the safe operating limits. This functionality may cause issues if coordinated movements are executed between two robots with different limits. The events are recorded and logged so the user can adopt the misconfigured software to comply within the safe operating limits.

The previously presented functionality is executed by the multiplexer nodes. As long as the software is configured to use the proper topics, the input switching is robust and does not allow unauthorized controllers to communicate with the hardware interface. This can however be bypassed by publishing directly into topics or actionservers provided by the hardware. This misconfiguration is also reported and logged so the users may fix the issues. This hazard can be reduced further by using the included firewall functionality for ensuring the access to the hardware interface cannot be used by any other node. Combining these two improves the fault tolerance of the robot as accidental and even malicious publishing to the hardware interface can be prevented.

The previously mentioned slowdown functionality is not perfect as discussed in the multiplexer implementation. The safety related speed control cannot actively slow down the ongoing motions on all of the provided message types when the system is commanded to slow down and the requirement 6 is not fully achieved. Only the messages with continuous command data work as intended. The event type messages, such as the FollowJointTrajectory, were investigated and had partially implemented functionality, before they were rejected. The slowdown functionality could not be reliable in every situation. Instead the interfaces complete the current motions with the old velocity and acceleration limits and the slower motions are only applied for new commands passed through the multiplexer nodes. To fix this more work needs

to be applied to the multiplexers with event based messages. However as recognized in the multiplexer section, the return on investment for the rework is not guaranteed and it requires more resources from the host system to operate correctly. The current implementations do not consume resources and response times are minimal for the event message based nodes.

To keep the resource consumption low, the velocity limits are only applied for joint angles. No forward kinematics are performed for the joint-chains and some interfaces do not even have feedback from the actuators. This may lead to the end effector exceeding the velocity limits, conflicting with the requirement 4. As mentioned in the multiplexer section this was seen to be excessive as the velocity limiting was added to catch extreme or badly misconfigured movements from passing into the hardware. The real protection should be executed by the fault monitors. These events are also reported and logged for the user so the issue can be resolved at the controller level.

Resource allocator provides the required services for the client controllers to request and relieve resources. It is easy to use and robust as it does not rely on internal memory to store the current state of the system. Querying the multiplexer nodes each time for normal queries slows down the response times for access requests, but the 1.5 seconds for a normal successful query was deemed acceptable as it is only done during control changes. Naturally the figure starts doubling when there are other requests queried on the system. The allocator processes only one request at a time. The override functionality provides the needed tool for recovering the robot from fault states. Using the normal methods only the controller on the resource can relieve it. When using the overrides the resources are handed over to the requesting controller and the original one is notified of the lost resources. The override also takes care of allocating any overlapping group that has been reserved on the nodes of the overridden target. This improves the safety of the system as a controller is not allowed to continue execution on the resource that may have been partially taken over by the overriding controller.

Overrides are intended to be used to resolve faults in the system. The provided functionality for the faults is simplified to a single message on a topic any of the components on the robot may connect to. Using dedicated monitoring components to detect and identify faults in the system, the information can be passed to other components. These may resolve the fault or forward it to the relevant party for inspection. The message also controls the multiplexer states slowing down the robot with the safety-related speed control or stop it with the protective stop. This is a category 2 stop state and it can be automatically resolved if the autonomous layer of the robot determines the fault to be safely recoverable. The thesis does not take account on how this could be done, but provides the functionality for executing the needed actions in practical manner. A possible method suitable for the system is referenced, but the full implementation did not belong into the scope of this thesis.

These faults can be monitored and manipulated from the HTTP interface. The faults are listed and color coded based on the actions the multiplexers take when the faults are received. This makes the identification of faults stopping or slowing down the

system effortless. The interface was also designated as the software E-Stop and it can be used from any computer or mobile phone connected to the same network as the robot. For researching the effects of simulated faults on the system the interface also includes the possibility of publishing faults from the interface. The same interface is also used to inspect the messages from the multiplexer nodes stated earlier. These are also color coded so different errors can be easily filtered from the listing. The interface is not perfect as only the numerical values of messages are used to denote the values instead of the human readable enumerations. As a secondary feature, the monitoring interface accomplishes the tasks required by showing and modifying the faults.

The testing on the actual Care-O-bot 4 proved the ROSGuard system to behave as in the simulated robot. The main issue on the real robot operation are the hardware faults. During the testing it was found out that if the arms were stopped from motion by the multiplexers, the robot hardware would issue faults on the joint actuators. In the simulated environment this did not occur and the only faults in the diagnostics were errors from faulty simulated 3d cameras on the torso. This combined with the observations on the firewall led to the problem that in certain modes the firewall would prevent the recovery functions on the hardware. In locked mode the firewall prohibits ROS service connections from forming on the nodes with the targeted topics. As a result the recovery commands using these services could not be issued to the hardware with locked topics. This was resolved by adding the protected mode into the firewall where the service connections were able to form.

The protected mode of the firewall is a stopgap solution for the problem with service connections. As discussed in the firewall section it is not perfect as it operates on polling principle and only updates the internode connection map periodically. This allows the publishers to deliver the message to the subscribers if the message is delivered in the timeframe the publisher has started advertising to the topic, but the firewall has not yet detected the connection. Workarounds for the issue are provided by using the persistent service connections, but the proper solution needs a complete rewrite how the firewall operates on protecting the subscriber nodes.

The testing on the selected use cases also brought out issues on the current safety standards. The standards require the motions of the robot to be stopped immediately using the emergency stop. In the selected case this could lead to further harm for the human interacting with the robot as stopping of the robot does not stop the hazard, but instead causes additional damages. When the robot is allowed to recover from the fault causing the hazards the risk of injury is decreased, assuming the recovery method was the correct response. The inherently safe design of the platform and tasks could limit the usability of the robot to levels, which could be unacceptable for normal user. The risk identification and assessment is critical for determining which actions the robot may take and to minimize the risks.

In conclusion the ROSGuard system presented meets the main requirements for it, the controller separation executed by the robust multiplexers. No error messages are generated by the components, when the system is configured properly as it

instructs the users on configuration of the system. The secondary requirements are also achieved to an agreeable level, not every interface provides the full functionality. The ease of use should also fulfill the requirements for the Intelligent robotics group, but it remains to be seen as the system is not yet in everyday use and the presented code examples are only for C/C++

References

[1] Intel chips timeline. URL https://www.intel.com/content/www/us/en/history/history-intel-chips-timeline-poster.html. Accessed: 2018-03-17.

[2] R. R. Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6): 52–59, 1997.

[3] Trends in the cost of computing. URL https://aiimpacts.org/trends-in-the-cost-of-computing/. Accessed: 2018-03-17.

[4] T. N. Theis and H.-S. P. Wong. The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017.

[5] M. S. Whittingham. History, evolution, and future status of energy storage. *Proceedings of the IEEE*, 100(Special Centennial Issue):1518–1534, 2012.

[6] F. Ingrand and M. Ghallab. Deliberation for autonomous robots: A survey. *Artificial Intelligence*, 247:10–44, 2017.

[7] D. Sidobre, X. Broquere, J. Mainprice, E. Burattini, A. Finzi, S. Rossi, and M. Staffa. Human–robot interaction. In *Advanced bimanual manipulation*, pages 123–172. Springer, 2012.

[8] ISO 13482:2014. Robots and robotic devices – safety requirements for personal care robots. Standard, International Organization for Standardization, Geneva, Switzerland, February 2014.

[9] M. Manti, A. Pratesi, E. Falotico, M. Cianchetti, and C. Laschi. Soft assistive robot for personal care of elderly people. In *Biomedical Robotics and Biomechatronics (BioRob), 2016 6th IEEE International Conference on*, pages 833–838. IEEE, 2016.

[10] E. Prassler, A. Ritter, C. Schaeffer, and P. Fiorini. A short history of cleaning robots. *Autonomous Robots*, 9(3):211–226, 2000.

[11] Karel Čapek. URL http://authorscalendar.info/capek.htm. Accessed: 2018-03-17.

[12] N. Davies. Can robots handle your healthcare? *Engineering & Technology*, 11 (9):58–61, 2016.

[13] G. Steinbauer. A survey about faults of robots used in robocup. In *RoboCup 2012: robot soccer world cup XVI*, pages 344–355. Springer, 2013.

[14] Intelligent robotics. URL http://eea.aalto.fi/en/research/intelligent_robotics/. Accessed: 2018-03-17.

[15] Fraunhofer IPA - Wir produzieren Zukunft - Fraunhofer IPA. URL https://www.ipa.fraunhofer.de/. Accessed: 2018-03-17.

[16] ROS.org | Powering the world's robots. URL http://www.ros.org/. Accessed: 2018-03-17.

[17] Access controls on topics and services? - ROS Answers: Open Source Q & A Forum. URL https://answers.ros.org/question/225577/access-controls-on-topics-and-services/. Accessed: 2018-04-14.

[18] SROS ROS Wiki. URL http://wiki.ros.org/SROS. Accessed: 2018-04-14.

[19] Trasport Security and ROS. URL http://wiki.ros.org/SROS/Tutorials/TrasportSecurityAndROS. Accessed: 2018-04-14.

[20] R. White, D. Christensen, I. Henrik, D. Quigley, et al. SROS: Securing ROS over the wire, in the graph, and through the kernel. *arXiv preprint arXiv:1611.07060*, 2016.

[21] Secure ROS. URL http://secure-ros.csl.sri.com/. Accessed: 2018-04-14.

[22] Secure ROS 0.9.2 documentation. URL https://sri-csl.github.io/secure_ros/. Accessed: 2018-04-14.

[23] Announcing Secure ROS - Discourse.ros.org. URL https://discourse.ros.org/t/announcing-secure-ros/1744. Accessed: 2018-04-14.

[24] ROS/Technical Overview - ROS Wiki. URL http://wiki.ros.org/ROS/Technical%20Overview. Accessed: 2018-03-04.

[25] TCP keepalive overview. URL http://tldp.org/HOWTO/TCP-Keepalive-HOWTO/overview.html. Accessed: 2018-03-04.

[26] R. Muthusamy and V. Kyrki. Decentralized approaches for cooperative grasp planning. In *Control Automation Robotics & Vision (ICARCV), 2014 13th International Conference on*, pages 693–698. IEEE, 2014.

[27] Names - ROS Wiki. URL http://wiki.ros.org/Names. Accessed: 2018-03-05.

[28] smach - ROS Wiki. URL http://wiki.ros.org/smach. Accessed: 2018-03-05.

[29] B. Povse, D. Koritnik, T. Bajd, and M. Munih. Correlation between impact-energy density and pain intensity during robot-man collision. In *Biomedical Robotics and Biomechatronics (BioRob), 2010 3rd IEEE RAS and EMBS International Conference on*, pages 179–183. IEEE, 2010.

[30] IEC 60204-1. Safety of machinery - electrical equipment of machines - part 1: General requirements. Standard, International Electrotechnical Commission, Geneva, Switzerland, October 2016.

[31] roscpp - ROS Wiki. URL http://wiki.ros.org/roscpp. Accessed: 2018-03-06.

[32] rospy/Overview/Services - ROS Wiki. URL http://wiki.ros.org/roscpp/Overview/Services#Persistent_Connections. Accessed: 2018-03-19.

[33] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[34] M. Vasic and A. Billard. Safety issues in human-robot interactions. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 197–204. IEEE, 2013.

[35] T. Malm, J. Viitaniemi, J. Latokartano, S. Lind, O. Venho-Ahonen, and J. Schabel. Safety of interactive robotics—learning from accidents. *International Journal of Social Robotics*, 2(3):221–227, 2010.

[36] ISO 13849-1. Safety of machinery – safety-related parts of control systems – part 1: General principles for design. Standard, International Organization for Standardization, Geneva, Switzerland, December 2015.

[37] D. Crestani, K. Godary-Dejean, and L. Lapierre. Enhancing fault tolerance of autonomous mobile robots. *Robotics and Autonomous Systems*, 68:140–155, 2015.

[38] joint_trajectory_controller/UnderstandingTrajectoryReplacement - ROS Wiki. URL `http://wiki.ros.org/joint_trajectory_controller/UnderstandingTrajectoryReplacement`. Accessed: 2018-01-29.

[39] roscpp: ros::NodeHandle Class Reference. URL `http://docs.ros.org/indigo/api/roscpp/html/classros_1_1NodeHandle.html#a0eae6a7fd8c216c3f0860aec44ca81ba`. Accessed: 2018-02-17.

[40] roscpp/Overview/Publishers and Subscribers - ROS Wiki. URL `http://wiki.ros.org/roscpp/Overview/Publishers%20and%20Subscribers/#Intraprocess_Publishing`. Accessed: 2018-01-20.

[41] netfilter/iptables FAQ: Questions about netfilter development. URL `https://netfilter.org/documentation/FAQ/netfilter-faq-4.html#ss4.5`. Accessed: 2018-01-30.

[42] ROS/Master_API - ROS Wiki. URL `http://wiki.ros.org/ROS/Master_API`. Accessed: 2018-01-30.

[43] "Schneier's Law" - Schneier on Security. URL `https://www.schneier.com/blog/archives/2011/04/schneiers_law.html`. Accessed: 2018-02-01.

[44] B. Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. John Wiley & Sons, 2007.

[45] async_web_server_cpp - ROS Wiki. URL `http://wiki.ros.org/async_web_server_cpp`. Accessed: 2018-03-04.

[46] care-o-bot - ROS Wiki. URL `http://wiki.ros.org/care-o-bot#Robots.2BAC8-Care-O-bot.Environment_variables`. Accessed: 2018-04-15.

[47] G. Bugmann and S. N. Copleston. What can a personal robot do for you? In *Conference Towards Autonomous Robotic Systems*, pages 360–371. Springer, 2011.

[48] MoveIt! Motion Planning Framework. URL `https://moveit.ros.org/`. Accessed: 2018-04-14.

[49] cob_command_gui - ROS Wiki. URL `http://wiki.ros.org/cob_command_gui`. Accessed: 2018-04-14.

[50] R. Oldenziel, A. A. de la Bruhèze, and O. De Wit. Europe's mediation junction: technology and consumer society in the 20th century. *History and Technology*, 21(1):107–139, 2005.

[51] L. Royakkers and R. van Est. A literature review on new robotics: automation from love to war. *International journal of social robotics*, 7(5):549–570, 2015.